FIG-FORTH

DEVELOPMENT LANGUAGE

INTERCEPTOR SOFTWARE

SUITABLE FOR THE

AMSTRAD

CPC-464

FIG-FORTH

TO THE READER

To the best of our knowledge, this manual is technically correct at the time of going to press. However, if you notice any mistakes or have any criticisms or suggestions then we would be grateful to receive them.

Yours Sincerely

Richard Paul Jones (Producer)

COPYRIGHT

WRITTEN BY BRIAN PERRY
PRODUCED BY RICHARD PAUL JONES

FOREWORD

This version of FORTH for the Amstrad CPC-464 computer is a complete and extended version of the FIG-FORTH standard as defined by the FORTH Interest Group. The extensions are in the form of WORDS not defined in FIG-FORTH but never the less regarded as necessary requirments to the language and special extensions to cover the special facilities offered by the AMSTRAD CPC-464 computer.

The manual is divided into several sections. Sections 1 and 2 are common to all users, whereas section 3 caters for the user who has had previous exposure to the FORTH language. Section 4 is an introductory guide to FORTH for the user with no previous experience in the use of the language. This section is not intended to be a full tutorial on the subject. It is intended to give an overview of FORTH and to allow the user to begin using the system as quickly as possible. For a complete indoctrination on the subject there are many excellent books available and appendix 5 should be consulted for recommended reading on this.

At all times the assumption is made where required, that the user is familiar with the operational aspects of the AMSTRAD CPC-464 and its BASIC language.

CONTENTS

INTRODUCTION

FORTH is not a new language. It was originally developed in the late sixties by Charles H. Moore, working on an IBM 1130 a "third generation" computer. The results he believed to be so powerful that he considered it a "fourth generation" computer language. The computer however would only allow five character identifiers so FOURTH became FORTH.

Since that time FORTH has grown in popularity both in the professional enviroment and in the area of the computer hobbyist.

Many so called "computer-buffs" shy away at the very mention of the word FORTH uttering crys of "REVERSE POLISH NOTATION", "LACK OF ERROR TRAPPING", "STACKS", "COME BACK BASIC, ALL IS FORGIVEN" etc. In essence, their problem is they have either never used FORTH or they will not use it because of fear of its unusual dialect. What still remains however is the undisputable fact that FORTH users are on the increase both in industry and in the home.

The author was introduced to FORTH some two years ago while reading an article on the subject. His interest grew after seeing a demonstration of FORTH being used to control a robot arm. The speed and flexibility of FORTH showed it to be an ideal language for the kind of work he was involved in (control engineering). Since that time he has used FORTH extensively in many projects that involved real time control.

FORTH is an unusual language. It casts aside many of the golden rules of programming. However as you begin to use it you will more than likely begin to see its power, and beauty. Be warned however, few people who learn FORTH ever go back to conventional languages.

LOADING FORTH

To load and run FORTH follow these simple steps.

1. Load the cassette into the recorder and rewind to the beginning of the tape.
2. Press the CTRL/SHIFT and ESC keys in that order to reset the computer.
3. Press play on the recorder.
4. Press the CTRL and small ENTER key on the keyboard followed by any key. FORTH will load and self start on successful completion.

The sign on message " Amstrad Fig-Forth Version 1.0 " will be displayed indicating the system is running.

It is suggested at this stage that a backup copy is made to safeguard against accidental damage to the master. To do this insert a blank cassette into the recorder and press play and record.

Type in the FORTH word SYSDUMP ensuring that you are in upper case characters and press the ENTER key. The FORTH system program will now be saved to cassette and can be reloaded as described above. For a further description of the SYSDUMP command see appendix 3.

USING FORTH

3.1 THE FORTH ENVIROMENT.

FORTH on the AMSTRAD CPC-464 occupies some 34K of memory. Fig 1a shows a memory map of the system. FORTH starts at 4000 hex, the immediate area above this contains the jump vectors for the COLD and WARM start operations and also the cold start parameters (see appendix II). Next is the precompiled or kernel vocabulary. This occupies around 10K bytes of memory and contains all FIG-FORTH words and extensions. The end of the dictionary contains a dummy word TASK which is purely to indicate the end of the precompiled portion. The free dictionary area begins immediately above TASK, the actual address can be found from the FORTH system variable HERE, although this value will change as new definitions are added to the vocabulary.

At a fixed 44 byte offset from HERE is the pad. This is a temporary text storage area for use by the system and the user. The start address for pad can be found by using the system variable PAD. Around 8K of free dictionary space is allocated in the initial system, which is more than enough for even the most intrepid programmer.

Address 8B00 hex sees the beginning of the terminal input buffer and the Parameter or Data stack. The parameter stack moves down towards HERE while the Terminal input buffer moves up towards high memory. The addresses of these can be found using the system variables TIB and S0 respectively. After a cold start these values will be the same.

Above the Terminal input buffer is the return stack and user variable area.
There are 40 hex bytes allocated for the user variables of which 30 hex bytes have been used. New user variables may be declared using an offset beginning with 32 hex, two bytes being used for each user variable defined.

Beginning at 8BE0 hex are eight block buffers, each buffer being made up of 2 header bytes, 128 data bytes and two tail bytes. These buffers are used in the reading and writing of data to and from the virtual memory disc system. The beginning and end address of the buffers can be found from the system constants FIRST and LIMIT respectively.

The virtual memory disc lies directly below the precompiled FORTH beginning at location 0400 hex. There are 14 edit screens available each screen being 1K bytes in length. For more information on the screens see section 3.8.

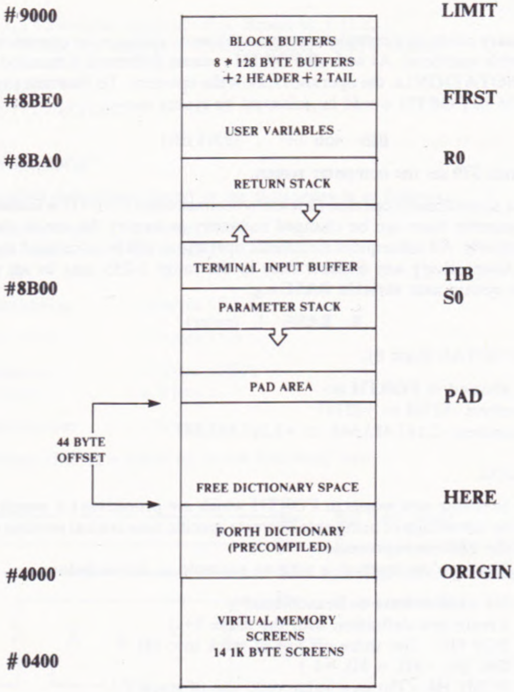


FIG 1.a FORTH MEMORY MAP

3.2 KEYBOARD INPUT.

All precompiled words in the FORTH dictionary have all been defined in upper-case characters, therefore ensure that the keyboard has upper-case selected before using the system. Any new definitions that are entered may be in upper or lower-case to suit the user.

All input from the keyboard is directed to the terminal input buffer prior to processing. A carriage return terminates entry from the keyboard and allows the inner interpreter to process the text held in the terminal input buffer. The buffer itself is 128 bytes long, its base address being found from the FORTH variable TIB.

The only editing facility available in the immediate mode is the (DEL) key. Should you make a typing error, backspace with the (DEL) key to the point where the error was made then retype the remainder of the line terminating of course with the enter key.

If you intend to type in a substantial ammount of text then the screen editor should be used, which is fully described in section 3.8

3.3 THE STACKS.

Forth uses two stacks in its operation, the data stack and the parameter stack. Both of these stacks operate on 16 bit values and are of the "last in first out variety". The parameter stack is realised by using the software stack of the Z80 processor and is used primarily for passing parameters between FORTH words. The return stack is software implemented and it is used mainly to temporarily store values during loop and branch operations and also by the inner interpreter to control program flow.

3.4 ARITHMETIC

The FORTH dictionary contains a complete integer arithmetic package for operation on 16 bit (single numbers) and 32 bit (double numbers). As with all FORTH systems arithmetic is handled using REVERSE POLISH or POST FIX NOTATION i.e. the operands follow the operator. To illustrate the point adding the two numbers 123 and 456 in FORTH would be achieved by typing in :-

```
123  456  +  .  (ENTER)
```

This would print the result 579 on the computer screen.

The number base for all arithmetic operations is set to decimal when FORTH is loaded from cassette or after a cold start. The number base can be changed to binary or hex by the use of the FORTH words BINARY or HEX respectively. All subsequent arithmetic operations will be calculated and displayed in the relevant number base. Alternatively any number base in the range 1-255 may be set up by storing the appropriate value in the system user variable BASE e.g.

```
8  BASE  !  (enter)
```

sets the number base to OCTAL (base 8).

The number range allowed in FORTH is:-
Single (16 bit) numbers -32768 to +32767
Double (32 bit) numbers -2,147,483,648, to +2,147,483,647

3.5 MACHINE CODE

It is relatively easy to define new words in FORTH which are primitives i.e. words written purely in machine code. This has the advantage of being able to create specific time critical sections of code which will run at the full speed of the Z80 microprocessor.

The easiest way to illustrate the method is with an example as shown below.

```
HEX             ( Set number base to hexadecimal )
CREATE 1+       ( Create new definition with the name 1+ )
   E1 C,        ( POP HL - Get value off top of stack into HL )
   23 C,        ( INC HL - HL = HL +1 )
   E5 C,        ( PUSH HL - Put new value onto top of stack )
   FD C, E9 C,  ( JP (IY) - Forces jump to inner interpreter )
SMUDGE          ( Allows new word to be "found" )
```

This example creates a primitive of the FORTH word 1+ which adds one to the value on the top of the data stack. The FORTH word C, encloses the byte preceeding it into the dictionary. When creating primitives in this way a few simple rules must be obeyed i.e.

a) If your routine uses the BC register pair save the contents before your routine is executed and retrieve them afterwards

b) Do not use the Z80 alternate register set. This may conflict with the AMSTRAD CPC-464 operating system.

c) Do not use the IY register, It is used by the system to point to the inner interpreter routine.

d) Always end your machine code with the JP (IY) instruction followed by SMUDGE.

3.6 RECURSION

Recursion is a technique by which a routine actually calls itself, a test usually being included in the routine to exit it. Recursion is not defined in Fig-Forth however this implementation supports it.

Usually in FORTH if a definition attempts to call itself it results in an error being generated. In this implementation the problem is removed by defining the word MYSELF which compiles the code field address of the word currently being defined into the definition. A good example of recursion is the calculation for the factorial function. This example calls itself repeatedly to calculate the factorial of a number in the range 0 to 7.

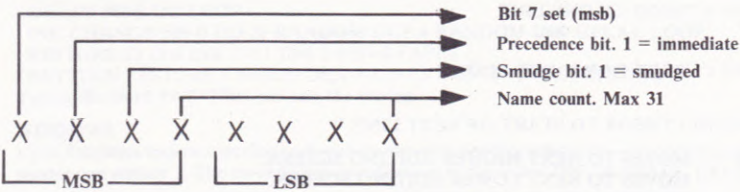
```
DECIMAL
: FACTORIAL ( Calculates factorial of n. Result to T.O.S. )
    DUP 0 < OVER 7 > OR IF ."Invalid number" QUIT THEN
    DUP 0=
    IF DROP 1
    ELSE DUP 1 - MYSELF *
    THEN ;
```

3.7 DICTIONARY FORMAT

The format of compiled definitions stored in the dictionary is as follows:-

Length of name with bit 7 set	(1 byte) NFA
Characters in name. Last character has bit 7 set	(n bytes)
Link to previous definition	(2 bytes) LFA
Pointer to execution code	(2 bytes) CFA
List of code addresses or	(n bytes) PFA
machine code if primitive	(n bytes)
Address of inner interpreter	(2 bytes)

The name field address (NFA) is made up in the following way:-



3.8 PRINTER

Under normal conditions all output is directed to the screen display. However using the FORTH word PRINTER will force all output to the centronics interface and thus the printer. Before using this word ensure that you have connected the printer and that it is on line. To revert back to normal screen mode the FORTH word SCREEN can be used to redirect all output back to the display.

The example below will dump the dictionary to the printer and return to screen output.

PRINTER VLIST SCREEN (enter)

3.9 SCREENS AND SCREEN EDITOR.

FORTH by convention uses editing screens of 16 lines by 64 characters with line numbers in the range 0 to 15. To comply with this the screen editor is implemented using MODE 2 text.

In a conventional disc based FORTH system, screens are held on disc and are called by the user as required. This implementation of FORTH simulates a disc by using a virtual memory system to hold screens in RAM. The reading and writing to the screens is totally transparent to the user. The words available to access the screens indirectly are fully described in appendix 4.

There are 14 edit screens avialable, numbered 1 to 14. Screens 0 is used to refer to keyboard input, therefore should not be used in this context.

To enter the screen editor, type 2 MODE and then n EDIT where n is the number of the screen you wish to edit. The display will clear and the line numbers 0 to 15 will appear together with the edit screen number. The text cursor which is always in the overwrite mode will be located at the beginning of line 0. Text may now be entered as required, any existing text at the cursor position being overwritten.

A number of editing commands are available these being described below.

EDITOR COMMANDS.

n EDIT - ENTERS EDITOR AT SCREEN n

CTRL/B - BLANKS ENTIRE SCREEN FILLING AREAS WITH SPACES

CTRL/O - OPENS UP A NEW LINE AT CURRENT CURSOR POSITION MOVING SUBSEQUENT LINES INCLUDING CURRENT LINE DOWN. ANY TEXT ON LINE 15 IS LOST.

CTRL/W - WIPES OUT THE LINE AT THE CURSOR POSITION MOVING SUBSEQUENT LINES UP. LINE 15 IS FILLED WITH SPACES.

CTRL/Z - ERASES THE LINE AT THE CURSOR POSITION FILLING IT WITH SPACES. NO OTHER LINES ARE AFFECTED.

CTRL/Q - QUITS THE EDITOR RETURNING TO THE INPUT MODE. ALL CHANGES MADE ARE MADE PERMANENT.

CURSOR CONTROL KEYS.

← MOVES CURSOR TO THE LEFT.

→ MOVES CURSOR TO THE RIGHT.

↑ MOVES CURSOR UP ONE LINE.

↓ MOVES CURSOR DOWN ONE LINE.

ENTER - MOVES CURSOR TO START OF NEXT LINE.

CTRL/→ **- MOVES TO NEXT HIGHER EDITING SCREEN.**
CTRL/← **- MOVES TO NEXT LOWER EDITING SCREEN.**
CLR **- INSERTS A SPACE AT CURRENT CURSOR POSITION AND MOVES ALL SUBSEQUENT TEXT TO THE RIGHT**
DEL **- DELETES THE CHARACTER AT THE CURRENT CURSOR POSITION MOVING SUBSEQUENT TEXT LEFT**

Screens may be moved outside of the editor with the FORTH word COPY e.g.

1 4 COPY

will copy screen 1 to screen 4. The original contents of screen 1 are not destroyed.

Screens may be compiled using n LOAD where n is the screen number to be compiled. If more than one screen requires compiling the FORTH word --> (pronounced arrow) may be used as the last word on any screen. This will force complilation of the next edit screen.

Edit screens may be saved to, and read from cassette using the FORTH words PUT and GET respectively. THESE words are illustrated below.

1 1 PUT (enter) - Saves screen 1 to cassette.

1 5 PUT (enter) - Saves screens 1 to 5 to cassette.

1 1 GET (enter) - Loads screen 1 from cassette

1 5 GET (enter) - Loads screen 1 to 5 from cassette.

In all cases ensure that the cassette recorder is set up for play or record prior to execution of the words as no prompt message is issued.

3.10 GRAPHICS AND EXTENSIONS

Many extensions are included in FORTH to cater for the colour and graphics facilities of the AMSTRAD CPC-464. Wherever possible the FORTH word used, has been choosen to be equivalent to its BASIC counterpart.

DRAWING & PLOTTING

The words provided for drawing and plotting are DRAW, DRAWR, PLOT and PLOTR. These are direct equivalents to the BASIC commands of the same name but require their X and Y parameters on the data stack prior to execution e.g:

```
: DRAW-LINE CLG 599 399 DRAW ;
```

will draw a line from the bottom left to the top right of the VDU screen in the current graphics pen. The graphics pen may be changed by using the FORTH word GPEN which again requires its parameter on the stack.

Here is an example which incorporates many of the graphics words available. It selects mode 1 and draws random lines on the screen in different colours. The inks are then changed at random to produce a pulsating picture. The program may be aborted by pressing the CTRL/Q keys.

```
: X1 599 RANDOM ; : X2 599 RANDOM ;
: Y1 399 RANDOM ; : Y2 399 RANDOM ;
: LINE X1 Y1 MOVE X2 Y2 DRAW ;
: 4-LINES 4 0 DO I GPEN LINE ;
: DRAW-LINES 50 0 DO 4-LINES LOOP ;
: DELAY 500 0 DO LOOP ;
: INK-CHANGE 500 0 DO 26 RANDOM DUP 4 RANDOM INK DELAY LOOP ;
: RESTORE 13 13 0 INK 0 0 1 INK 1 PEN 0 PAPER ;
: PATTERN RESTORE 1 MODE DRAW-LINES INK-CHANGE RESTORE ;
```

Typing the word PATTERN will run the routine.

WINDOWS

Up to 8 streams may be specified each with its own window characteristics. The following example sets up a window on stream 3. The co-ordinates specified are in the order TOP, BOTTOM, LEFT, and RIGHT.

```
3 STREAM ( selects text screen )
10 20 15 60 WINDOW ( Set up window ).
VLIST ( Shows extent of window ).
0 STREAM ( RETURN TO DEFAULT SCREEN )
```

USER DEFINED CHARACTERS

FORTH provides two words for creating user defined characters. The first symbol is equivalent to BASICS SYMBOL AFTER COMMAND e.g.

```
128 SYMBOL
```

will allow all characters including and after 128 to be defined. The word CHARACTER allows the re-definement e.g.

```
HEX
0F 03 05 09 10 20 40 80 A8 CHARACTER
```

will redefine character A8 (decimal 168) from the $^{1}/_{4}$ symbol to a diagonal arrow.
To print this character we use the FORTH word EMIT e.g.

```
HEX A8 EMIT
```

PENS AND INKS

Pens and Inks may be changed within the constraints of the current screen mode as shown below.

```
1 MODE ( Select mode 1 )
3 PEN ( Select pen 3 ie red in defualt )
13 13 0 INK ( Change pen 0 to ink 13 )
16 9 1 INK ( Change pen 1 to inks 16 and 9 i.e Flashing ).
```

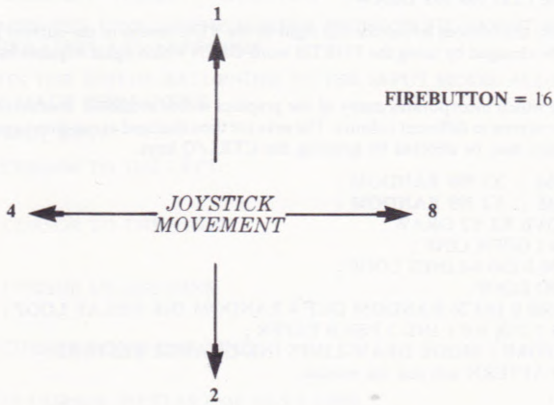
GRAPHICS TESTING

Graphics points may be tested with the TEST and TESTR commands. These return to the top to the stack

the pen number at the X and Y co-ordinates supplied as parameters e.g.
CLG 100 100 PLOT (PLOTS POINT IN CURRENT GPEN)
100 100 TEST (RETURNS PEN TO TOP OF STACK)

JOYSTICKS

Two commands ie JOY0, JOY1 returns the value of the joystick as the second stack entry with a true flag as the top. If the joysticks are not active a false flag is placed on the top of the stack only. The values returned for an active joystick are shown below.



TIMER

One timing facility is available in FORTH. The word 0TIME will reset the timer to zero and begin the count sequence. The word ?TIME will put the current timer value onto the stack as a double number. The timer is incremented in steps of $^1/_{300}$ of a second. The timer may be used amongst other things for timing sections of code e.g.

0TIME ."HELLO" ?TIME D.

Will print the number 12 indicating that the print statement took $12 \times {}^1/_{300}$ seconds i.e. approximately 0.04 seconds.

ARRAYS AND TEXT STRINGS

3.11

Single and double dimension arrays may be created using the defining word 1-ARRAY and 2-ARRAY respectively e.g.

10 1-ARRAY FRED

will create a single dimension array of 10 elements called FRED. Note that there are in fact 11 elements in the array as they range from 0-10 (this is the same as the BASIC version of arrays). Below is an example which creates a two dimension array of 3 rows by 4 columns with the name of BILL.

3 4 2-ARRAY BILL

To write a value into for example, the array BILL, we specify first the number to store, then the row and column numbers and then the array name followed by the FORTH word ! which stores the number e.g.

235 1 4 BILL !

will store the number 235 at row 1 column 4 of the array BILL.

Reading from the array is very similar. We specify first the row and column numbers then the array name then the FORTH word @ (FETCH) e.g.

1 4 BILL @

will read row 1 column 4 of the array BILL and store the value on the top of the stack.

Although the examples shown are for double dimension arrays the same format applies to single dimension arrays. Note however that no tests are performed by FORTH to check that you are accessing a valid array location therefore unpredictable results may occur if this situation arises.

TEXT STRINGS

In addtion to the normal text handling words provided in FORTH, two special words have been provided to help in text string handling.
The first word is STRING and is a defining word used in the form as shown in the example below.

20 STRING NAME

This creates a dictionary entry called NAME and allocates space for up to 20 text characters. Executing the word NAME after compilation places the address of the string stored in NAME onto the stack where it may be printed by the FORTH word TYPE.

The word INPUT$ allows a string to be entered from the keyboard terminated by a carriage return and stores the string at the address preceding the word. The example below shows how the words STRING and INPUT$ are used.

```
30 STRING SURNAME ( creates header called SURNAME and allocates 30 spaces )

: ASK-NAME
    CLS CR ." WHAT IS YOUR SURNAME PLEASE"
    SURNAME INPUT$ ( Read characters from keyboard and store at SURNAME )
    ." YOUR SURNAME IS " SURNAME TYPE CR ; ( Print surname on screen )
```

SOUND GENERATION

3.12

The approach to sound generation using FORTH is somewhat different to that employed by BASIC. The method adopted is to create named sound data blocks which hold the various parameters required for sound generation. These data block can be played using the FORTH word PLAY. The defining word used to create the data blocks is called SOUND and the general form of SOUND is as shown below.

478 1 SOUND BEEP

This creates a sound name BEEP with a tone period of 478 and to be executed on channel A of the sound generator. Typing the following will cause the sound to be played.

BEEP PLAY

The word PLAY tries to add the referenced sound to the sound queues of of the relevant channel (there are five queues per channel). If an attempt to add a sound to the queues failed an error message is given and the program aborts.

The layout of the sound data blocks is shown below. Their exact useage being fully described in the CPC-464 OPERATING SYSTEM FIRMWARE SPECIFICATION (SOFT 158).

BYTE 0	Channels to use and rendezvous requirments.
BYTE 1	Amplitude envelope (default 0)
BYTE 2	Tone envelope (default 0)
BYTE 3	Tone envolope (default 0)
BYTES 3-4	Tone period in the range 0-4095 (0 is no sound)
BYTE 5	Noise period in the range 0-31 (0 is no sound)
BYTE 6	Initial amplitude (default 4)
BYTES 7-8	Duration or envelope repeat count (default 20)

The tone periods and there corresponding musical notes can be determined from the table provided in Appendix VII of the CPC-464 user instructions. The channels to use A, B and C are represented by the numbers 1, 2 and 4 respectively.

The parameters of the sound block may easily be modified to build up a complex sound. The words provided to enable this to be carried out are:-

1). CHANNEL

Used in the form 2 BEEP CHANNEL now assigns the sound BEEP to Channel 2. A number 3 would assign it to sound channels 1 and 2 simultaneously etc.

2). AMP-ENV

Use in the form (n) BEEP AMP-ENV will assign a predefined amplitude envelope where (n) is in the range 1-15 (see later).

3). TONE-ENV

Used as per AMP-ENV but specifying a tone envelope (see later).

4). PERIOD

Used in the form 239 BEEP PERIOD will change the note played from a middle C to a C one octave higher on the musical scale. The number supplied must be in the range 1 to 4095. Again refer to the CPC-464 user instruction manual for the period to musical note relationship.

5). NOISE

A value supplied in the form (n) BEEP NOISE will add noise in the sound of BEEP. The number (n) must lie in the range 0-31 where 0 represents no noise.

6). VOLUME

Used in the form (n) BEEP VOLUME will set the volume level to the value (n). The value n must lie in the range 0 to 15 where 0 is no volume and 15 is maximium.

7). DURATION

Used in the form (n) BEEP DURATION where (n) is in the range -32768 to +32767. If (n) is a positive number then that represents the duration of the note in 1/100ths second. Where the number is negative, the positive value of this number represents the number of times the volume envelope (if specified) should be repeated. A value of 0 causes the duration to be governed by the amplitude envelope supplied.

8). SHOLD

This when used in the form BEEP SHOLD will prevent that sound from running until released (see RELEASE).

9). SFLUSH

This is used in the form BEEP SFLUSH and sets the flush bit of the sound block. This forces the sound queues to abandon any current sound enabling the referenced sound to play immediately.

10). RELEASE

This is used in the form (n) RELEASE and will release a sound on channel (n) which was previously held using SHOLD.

11). FREEZE

This will stop all sounds in mid-flight.

12). CONTINUE

Releases sounds which have been stopped using FREEZE.

13). RESET

This re-initialises the sound manager and sound chip and clears all queues

14). SWAIT

Used in the form (n) SWAIT will cause the program to wait until there is room on a sound queue of channel (n) before adding any more sounds to the queue.

15). SQ

Used in the form (n) SQ will return to the top of the data stack the status of sound channel (n). The number returned is encoded a follows.

Bits 0..2 Number of free slots in the sound queue.
Bits 3 Channel is waiting to rendezvous with channel A.
Bit 4 Channel is waiting to rendezvous with channel B.
Bit 5 Channel is waiting to rendezvous with channel C.
Bit 6 The channel is held.
Bit 7 The channel is producing a sound.

ENVELOPES

Both named amplitude and tone envelopes may be created with the FORTH words ENV and ENT. The envelopes are of a similar construction to those used in BASIC, each envelope having up to five sections of three parameters, the main difference being that the parameters are all in reverse order to that specified by BASIC.

The example below shows the creation of an amplitude envelope of the general form shown in Fig 3a.

The main point to note is that the envelope parameters are reversed i.e. the last envelope section is declared first. The order for each section of the envelope is pause time, step size and step count. The number 4 before the word ENV is to indicate the number of sections in the envelope.

The above method for creating amplitude applies in exactly the same way to tone envelopes using the ENT word.

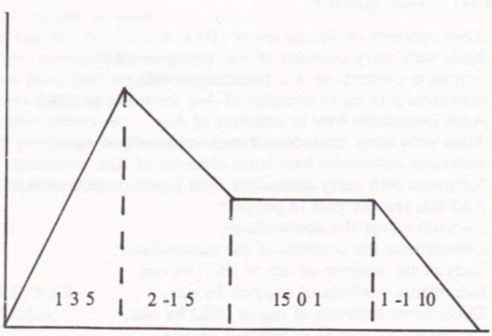


Fig 3a waveform for SHAPE

```
DECIMAL
1 -1 10  15 0 1  2 -1 5  1 3 5 4 ENV SHAPE ( create envelope )

1 SHAPE                 ( assign SHAPE to envelope number 1 )
100 1 SOUND GONG        ( create sound name GONG )
0 GONG DURATION         ( allow envelope to control length )
0 GONG VOLUME           ( Allow envelope to control volume )
1 GONG AMP-ENV          ( Assign envelope 1 to GONG )
GONG PLAY               ( Play it )
```

ASSEMBLER

A FORTH Assembler is very different to the standard assembler most of us are used to. It is designed to allow easy creation of new FORTH machine code primitives using easy to understand mnemonics. The target memory for the assembled code is the free dictionary area (designated by the FORTH word HERE). No symbol table is used by the assembler, instead all branch or jump references are handled by high level structures via the stack.

The actual mnemonics used are in fact a mixture of Z80 AND 8080 mnemonics (the 8080 is the forefather of the Z80) the reason for using the 8080 mnemonics being that they tend to resolve some of the addressing problems that occur in similarly named Z80 instructions e.g. several forms of addressing modes for the Z80 LD instruction.

One last point which must be pointed out before we examine the assembler in detail is that the normal FORTH post fix notation still applies between opcodes and operands used in the code (see later).

INSTRUCTION SET.

The mnemonic instruction set supported by the FORTH assembler is given below together with a brief description of their functions. For a full description it is suggested that the user consult one of the many books available relating to the Z80 microprocessor.

a). ARITHMETIC AND LOGIC GROUP.

ADD	Adds contents of Acc to reg or (HL).
ADC	Adds with carry contents of Acc to reg or (HL).
SUB	Subtracts contents of Acc from reg or (HL).
SBC	Subtracts with carry contents of Acc from reg or (HL).
ADI	Adds immediate byte to contents of Acc.
ACI	Adds with carry immediate byte to contents of Acc.
SUI	Subtracts immediate byte from contents of Acc.
SBI	Subtracts with carry immediate byte from contents of Acc.
DAD	Add HL register pair to reg-pair.
DAA	Decimal adjust the accumulator.
CPL	Compliment the contents of the accumulator.
INC	Increments contents of reg of (HL) by one.
INX	Increments contents of reg-pair by one.
DEC	Decrements contents of reg or (HL) by one.
DCX	Decrements contents of reg-pair by one.
OR	Logical OR accumlator with reg or (HL)
ORI	Logical OR immediate byte with accumulator.
XOR	Exclusive or accumulator with reg or (HL).
XRI	Exclusive or accumulator with immediate byte.
AND	accumulator with reg or (HL)
ANI	AND accumulator with immediate byte.
CP	Compare accumulator with reg or (HL).
CPI	Compare accumulator with immediate byte.
SCF	Set carry flag.
CCF	Clear carry flag.

b). LOAD AND STORE GROUP

STAX	Load D (BC) or (DE) from accumulator.
LDA	Load accumulator from (addr).
LDAX	Load accumulator from (BC) or (DE).
LHLD	Load HL from (addr).
XCHG	Exchange contents of DE and HL.
XTHL	Exchange (SP) and HL.
STA	Load (addr) from accumulator
SHLD	Load (addr) from HL.
SPHL	Load SP from HL.
LXI	Load reg-pair with 16 bit immediate data.
MOV	Load reg or (HL) from reg or (HL)

Note that (HL) in the FORTH assembler is designated by the word M.

c). CALL AND RETURN GROUP.

RET-C	Return if carry flag set.
RET-NC	Return if carry not set.
RET-Z	Return if result zero.
RET-NZ	Return if result not zero.
RET-P	Return if result positive.
RET-M	Return if result negative.
RET-PO	Return if parity odd.
RET-PE	Return if parity even.
CALL	Call subroutine at addr.
JP	Jump to address.

d). ROTATE AND SHIFT GROUP.

RLA	Rotate accumulator left through carry.
RRA	Rotate accumulator right through carry.
RLCA	Rotate accumulator left circular.
RRCA	Rotate accumulator right circular.

e). MISCELLANEOUS.

PUSH	Push reg-pair to stack.
POP	Pop reg-pair from stack.
OUT	Output accumulator to port.
IN	Input from port to accumulator.
EI	Enable interrupts.
DI	Disable interrupts.
HALT	Halt processor until reset or interrupt.
NOP	No operation.
RST	Call a restart operation.

f). STRUCTURES

BEGIN — AGAIN
BEGIN — UNTIL
BEGIN — WHILE — REPEAT
IF — ELSE — THEN

g). CONDITIONALS

0=	Tests for the zero flag being set.
CS	Tests for the carry flag being set.
PE	Tests for even parity.
0<	Test for sign flag being set i.e. (minus)
NOT	Negates the above conditionals.

USING THE ASSEMBLER

The assembler is invoked by using the FORTH word CODE which creates a dictionary entry with the name following CODE and then assembles the mnemonics following. The example given below shows the use of the assembler in its simplest form.

```
CODE ADD ( nl n2 — n3 as sum of nl + n2 )
        HL POP ( Get 1st number from stack )
        DE POP ( Get 2nd number from stack )
        DE DAD ( Add HL and DE result in HL )
        HL PUSH ( Put result ot top of stack )
        NEXT ( Jump to inner interpreter )
C;      ( End of definition return to FORTH )
```

In the above example the word CODE creates a new dictionary header with the name ADD. Note that the mnemonics that follow have their operands preceeding the opcode which is the reverse of a standard assembler. The word NEXT compiles the JP (IY) instruction into the definition which forces a jump to the inner interpreter, THIS

MUST ALWAYS BE INCLUDED AS THE LAST WORD OF A DEFINITION. The last word C; completes the assembly process and returns to normal input mode. If the definition was created using the screen editor it may now be compiled using LOAD, and tested.

The second example shown below takes two numbers from the stack and logically OR's them, the result being placed on the stack.

```
CODE ORR ( n1 n2 — n3 as logical OR of n1 and n2 )
        HL POP ( Get 1st number )
        DE POP ( Get 2nd number )
        L A MOV ( Load A from L )
        E OR ( OR accumulator with E )
        A L MOV ( Put result in L )
        H A MOV ( Load A from H )
        D OR ( OR accumulator with D )
        A H MOV ( Put result in H )
        HL PUSH ( Put 16 bit result to stack )
        NEXT ( Jump to inner interpreter )
C; ( End definition )
```

Note the register useage with 8 bit loads i.e. the source register is first, followed by the destination register e.g. L A MOV where L is the source and A is the destination.

The last example shows the use of the built in structures which are available as standard in the assembler. These are the BEGIN — WHILE — UNTIL — etc. and the IF — ELSE — THEN. These structures take care of any relative jumps that would normally be written in assembly code. To aid using them a number of coditional tests are provided these having already been detailed above. The example below defines a word named NEWFILL which fills an area of memory with a specified byte.

```
CODE NEWFILL ( addr n b — )
   C L MOV ( Save IP into HL )
   B H MOV
   DE POP ( Get byte to fill with )
   BC POP ( Get count of bytes )
   XTHL ( Exchange (SP) and HL )
   XCHG ( Exchange DE and HL )
BEGIN ( Start of NEWFILL loop )
   B A MOV C OR ( Check for count zero )
   0= NOT ( Test for zero flag not set )
   WHILE ( While flag is not zero )
        L A MOV DE STAX ( Store byte into address )
        DE INX BC DCX ( Incermant address and decrement count )
   REPEAT ( Go back to begin )
   BC POP ( Restore IP from stack )
NEXT C; ( Jump to inner interpreter and exit assembler )
```

Subroutines may be defined if required (although in FORTH this is not really necessary) using the word LABEL in the form :-

LABEL name (code mnemonics)

and may be called using the assembler CALL instructions e.g.

name CALL

Note that the ASSEMBLER vocabulary must be selected to use LABEL.

INTRODUCING FORTH.

4.1 WHAT IS FORTH

FORTH is a somewhat unique language. It has been described by its creator as an operating system, high level language, a set of development tools and a software design philosophy.

FORTH is fast when compared to interpreted languages such as BASIC, on average running ten to twenty times faster. It is a very transportable language in that it adheres to well defined standards. A FORTH program which was designed to run on one type of computer can usually be tansferred to a totally different machine with very little modification, unlike BASIC which has many different dialects.

To write a program in FORTH you define new commands, or to give them their correct title WORDS, in terms of WORDS which exist in the main core or VOCABULARY of the language. This approach gives FORTH the advantage in that rather than actually writing a program you are in fact extending the language. Using this approach you can create new control structures to add to, for instance the compiler.

FORTH can be considered to consist of four main parts:-

a). The dictionary.
b). The keyboard interpreter.
c). The stacks
d). The disc.

4.2 THE DICTIONARY.

With the FORTH system up and running (see section 2) type in the FORTH word VLIST and press the enter key (from now on denoted by (enter)). Ensure that the characters you type in are in upper case only as FORTH will not understand them in lower case. You shouid see the screen fill up with many strange words. These words make up the FORTH dictionary. Each of the words can be thought of as a command which when typed in executes a specific routine. For example the word VLIST executes a routine which prints the dictionary to the screen. If you examine this list closely you will see that the word VLIST is also included.

Now type in the word COLD followed by enter. The screen will clear and the original sign on message will appear. The word COLD executes a routine which causes the system to reboot itself as if it had just been loaded in from cassette.

A complete list of these words and their meaning is given in appendix 3 and 4 but do not try to absorb all of these at once, familiarity will come with time. More than one word can be typed in at a time but they must be separated by at least one space e.g.

VLIST VLIST VLIST (enter)

will cause the dictionary to print three times in succession. The listing can be exited in mid stream by pressing CTRL/Q. Notice at the end of the listing the letters OK. appear. This is to let you know that FORTH has executed all of the instructions you requested and is now waiting for further input.

Should you at any time get an error message, this will more than likely signify that you have made a typing error, just retype in the commands taking care with spelling, spaces etc.

4.3 THE KEYBOARD INTERPRETER.

When you type in characters on the keyboard, as well as appearing on the screen, they are sent to an area of memory known as the terminal input buffer. When you press the enter key a routine known as the keyboard interpreter, searches through the dictionary and attempts to match up the words in the terminal input buffer with the words in the dictionary. If it is successful it executes the routine associated with the words, if it is unsuccessful it presumes that what was typed in was a number and therefore attempts to convert the characters to a binary number. If this fails then the interpreter issues an error message (usually error 0) which means the interpreter has not understood what was typed in. Try typing some random characters into the computer to illustrate the error message. FORTH should echo whatever was entered via the keyboard followed by the message error 0.

Now type in the following example being careful to insert spaces between each FORTH word. Remember the phrase (enter) means press the enter key, don't try and type it in.

```
: WELCOME CLS ."Welcome to Fig-Forth" CR ; ( enter )
```

If all went well you should get an OK prompt back, if you got an error message instead type in cold to

reboot the system and start again. Now type in the word WELCOME and press enter. Hey-presto, the screen clears and the message "Welcome to Fig-Forth" appears with the OK prompt on the next line. If you now examine the dictionary by typing VLIST you will see that WELCOME has been added at the top. We have in fact created our first FORTH program or definition. So what exactly happened?

After we had typed in the line of text and pressed enter, the keyboard interpreter began to execute the FORTH words in the line. The first word it encountered was the : (pronounced colon). This word tells the interpreter that we wish to create a new definition called WELCOME. At this point FORTH switches to compile mode and enters the name of the new word onto the dictionary. The words following WELCOME are compiled onto the dictionary after WELCOME, as the execution or run time part of the defintion. The FORTH word ; (semi-colon) stops compilation and returns to the normal execution mode of the interpreter. We have introduced a few new FORTH words in this example the first, CLS, is exactly the same as the BASIC command and simply clears the screen. The words ." and " (pronounced dot quote and quote respectively) are equivalent to the BASIC PRINT " " command. Any text between . " and " will be printed onto the screen. It is important to note that a space must be included between the ." and the first character of the text though this space is not actually printed.

4.4 THE STACK.

All data that is processed by FORTH is done via an area of memory known as the data or parameter stack. This stack can be likened to a stack of coins where the last item put on the stack is on the top. This type of arrangement is called a last in first out stack or L.I.F.O. stack for short. To illustrate the action of the stack type in the following:-

1 2 3 4 5 (enter)

FORTH should respond with the OK prompt. Now type the FORTH word . (pronounced "dot") and press enter, the number 5 will be printed. Type . and enter 4 more times and the numbers 4 3 2 1 should appear. What we have done is to initially put five numbers onto the stack and then retrieve them one by one. Fig 2a shows the stack after we have typed in the five numbers and after we have retrieved and printed the first two.

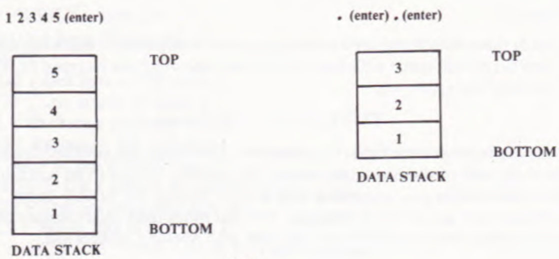


Fig 2a

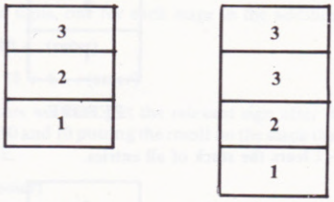
The important point to remember here, is that the numbers on the stack are destructively removed NOT copied as we might have expected. If we therefore have a number on the stack which we need to use twice, or even more times, we need some method of preserving it for future use. Fortunately FORTH provides us with a WORD which allows us to do just that. Type in the following:-

SP! 1 2 3 DUP (enter) .S (enter)

You should see the numbers 1 2 3 3 appear on the screen.
We have introduced some new FORTH words here, so lets examine them one by one. The FORTH word SP! simply clears the stack of all current entries, the FORTH word DUP duplicates the top entry on the stack (exactly what we were looking for) and the word .S prints out all the numbers on the stack without destroying them. This command is particularly useful when you wish to see the state of the stack without removing any values.

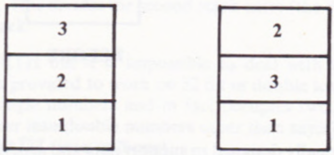
FORTH provides a number of words for manipulating the contents of the data stack. The major ones are described below in diagramatic form with a short description of their function.

i) DUP - Copies the top entry on the stack.



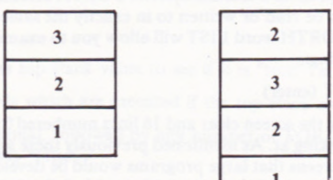
BEFORE AFTER

ii) SWAP - Swaps the top two stack entries.



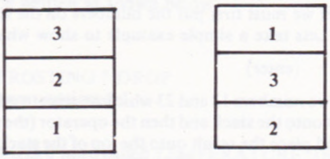
BEFORE AFTER

iii) OVER - The 2nd stack entry is copied to the top.



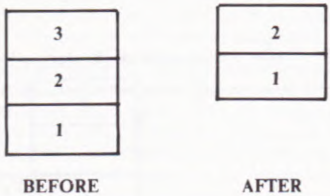
BEFORE AFTER

iv)
ROT - The 3rd stack entry is removed to the top.

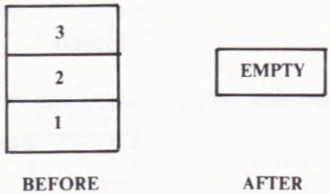


BEFORE AFTER

v) **DROP - Removes the top stack entry.**



vi) **SP! - Clears the stack of all entries.**



4.5 THE DISC.

Forth was originally designed to support data and program storage on disc. Programs are traditionally stored on disc in 1K byte blocks or screens made up of 16 lines of 64 characters. These screens could be called from disc as required and stored by FORTH in an area of RAM known as the block buffers. Sufficient block buffers were allocated to store only two or three screens of data at any one time. This did not pose a problem for large programs as the process of reading and writing screens to and from disc was very fast and was also transparent to the user, access being carried out under control of the FORTH system.

The AMSTRAD CPC-464 version of FORTH does not support a real disc system. Instead it allocates 14K of memory to pretend it is a disc system. FORTH screens are stored in this area of memory (14 screens maximum) and can be read or written to in exactly the same way as would be done in a true FORTH disc enviroment. The FORTH word LIST will allow you to examine the contents of any of the screens available. Type in the following:-

1 LIST (enter)

You should see the screen clear and 16 lines numbered 0 to 15 will be displayed together with the screen number you are looking at. As mentioned previously these screens are organised as 16 lines of 64 characters and it is on these screens that large programs would be developed and tested. To allow text to be placed on the screens, a full screen editor is available and is fully described in section 3.9.

4.6 NUMBERS AND ARITHMETIC

FORTH handles numbers in a somewhat unusual way . To start with it can only deal with integers usually in the range of -32767 to +32768 (16 bit signed numbers) though it can deal with larger integers which we will deal with later. Its second unusual point is the way we write arithmetic in FORTH. To add two numbers in FORTH we must first put the numbers on the stack and then tell FORTH what we wish to do with the numbers. Lets take a simple example to show what we mean.

12 23 + . (enter)

Here we have two numbers 12 and 23 which we wish to add together and then print the result. First of all the numbers are put onto the stack and then the operator (the plus sign) tells FORTH to add together the top two stack entries and place the result onto the top of the stack. The FORTH word . (dot) prints the top stack entry which in this case is our result. This method of handling numbers, where the operator comes after the numbers is known as POST FIX or REVERSE POLISH NOTATION and is fundamental to all FORTH arithmetic. Lets try some more simple examples to familiarise ourselves with this method of working. Here

the normal method of writing the expression is shown on the left with the equivalent FORTH expression on the right. Type in the examples making sure that they work and then try to see exactly what is happening to the stack. To help refer to appendix 4 for a full list and description of the operators.

24 + 12 + 10 = 46 **24 12 10 + + . (enter)**

In the above example note there are two plus signs, one for each stage in the addition.

132 - 10 = 122 **132 10 - . (enter)**

50 + 10 - 6 = 54 **50 10 + 6 - . (enter)**

In this example where we have mixed operators we must put the relevant sign after the numbers we are operating on. The first part adds the numbers 50 and 10 putting the result on the stack then we subtract the 6 leaving the final result on the top of the stack.

5 * 6 = 30 **5 6 * . (enter)**

Here we are using the FORTH word * which multiplies the top two numbers on the stack leaving the result on the stack.

30 / 5 = 6 **30 5 / . (enter)**

This is the division operator used in FORTH. This divides the second stack entry (the 30) by the top entry (the 5) leaving the result on the top of the stack.

There are many other operators in FORTH but it is impossible to deal with them in this short introduction. In particular there are operators provided to work on 32 bit or double length numbers. One double length number is equivalent to two single numbers and in fact occupies two stack positions in memory. It is not intended to delve any further into double numbers other than saying that they greatly increase the number handling capacity of FORTH (into millions).

The only satisfactory way to become aquainted with FORTH number notation is practice. Try examples of your own checking the results as you proceed creating more complex expressions as you master the simple ones. If you have difficulty it is suggested that you purchase one of the many FORTH tutorials on the market (see appendix 5).

4.7 MAKING DECISIONS

FORTH supports a structure which is very similar to the IF statement used in BASIC.
This is the IF - - ELSE - - THEN statement.
The general form of this in FORTH is:

1). The word IF which checks the top stack value to see if it is "true" i.e. a non zero value.

2). Any number of FORTH words which are executed if the top stack entry was true.

3). The word ELSE which is optional, followed by any number of FORTH words that are executed if the top stack entry found by IF was "false".

4). The FORTH word THEN which terminates the expression. This must be included unlike some BASICS which allow this part to be ommited.

The use of this can be best illustrated by an example. Lets say we have an electronic circuit connected to our computer which monitors the temperature of a refrigerator. We need to display on the screen a warning message if the temperature inside the fridge ever rises above zero degrees centigrade. The particular part of the FORTH program which does this could be written as shown below.

```
: TEMPERATURE
DUP 0>
IF ." WARNING FRIDGE DEFROSTING " DROP
ELSE ." Temperature is " . ."degrees centigrade"
THEN;
```

As this is part of a main program it is defined as a word called TEMPERATURE. The colon as we have already mentioned starts the compile mode within FORTH to create a new defintion. The DUP statement duplicates the top stack entry which in this case we presume to be the refrigerator temperature put there by some other FORTH word which actually did the measuring. The FORTH word 0>checks the top stack

entry (removing it in the process) to see if it is greater than zero. If it is then the test was "true" and a 1 is placed on the top of the stack. If it is zero or less than, then the test leaves a "false" or 0 on the stack.

The IF takes this stack value and IF IT IS TRUE it prints the warning message on the screen. We then DROP the duplicated temperature value from the stack as we no longer require it. IF IT IS FALSE the warning message and DROP are ignored and the ELSE part is executed which prints the temperature message. Note the . (dot) between the two print statements, this removes the duplicated temperature value from the stack and prints it on the screen. The THEN terminates the IF structure and the semi-colon completes the definition.

Type in the above example using the screen editor, compile it and then try it using different values, positive and negative, on the stack as shown below.

10 TEMPERATURE (enter) will print the warning message.
-5 TEMPERATURE (enter) will print the temperature.

4.8 REPETITION.

FORTH supports several methods for repeating a sequence of events. The first of these is the DO LOOP and is an equivalent of the BASIC FOR NEXT loop. Type in the following example an run it You should see the numbers 1 to 10 on the screen.

```
: TEST1 11 1 DO I . LOOP ;
```

This routine we have called TEST1. The numbers 11 to 1 represent the start and end values of the loop. The loop starts with a value of 1 and executes all code between the words DO and LOOP, incrementing the loop value at each pass until it reaches the limit, which in this case is 11. Note that when the limit is reached, the loop falls through i.e. it is not executed for the 11th time. The FORTH word I places the current loop index onto the stack where it can be printed using the FORTH word . (dot). This gives us our count from 1 to 10 on the screen.

The loop can be made to count in increments of more than one step using the FORTH word +LOOP (equivalent to the BASIC STEP command) as shown below.

```
: TEST2 11 1 DO I . 2 +LOOP ;
```

In this example the value of 2 is added to the loop index to force the loop to increment by 2. In this case the loop ends at the count of 9 as the next value would equal (or exceed depending on the index) the loop limit. The loop can be made to count downwards as shown in example TEST3.

```
: TEST3 1 11 DO I . -1 +LOOP ;
```

Note that the loop limits are reversed compared to the previous examples and that the index is incremented in steps of -1. Finally we have an example of nesting more than one DO LOOP in a routine.

```
: SQUARE CLS 250 150 ORIGIN
100 0
DO 100 0
    DO I J PLOT
    LOOP
LOOP ;
```

This example which we have called SQUARE plots a square onto the screen using the the graphics extensions supplied as part of the Fig-Forth program. There are several new words introduced here which may need some explanation.

The first word is ORIGIN which is equivalent to the BASIC command of the same name ORIGIN takes two numbers from the stack, in this case 250 and 150 and locates the graphics cursor at the x and y co-ordinates respectively.

The word PLOT lights up a pixel on the screen at the x and y co-ordinates given by the top two numbers on the stack. In this example these co-ordinates are supplied by the FORTH words I and J which place the respective loop indexes onto the stack.

As we have already mentioned, this routine plots a square 100 pixels by 100 pixels onto the screen. The time taken to plot all of the pixels (10000 of them) is approximately 4.5 seconds. An equivalent routine written in Basic takes around 24 seconds. This gives FORTH a 5 fold speed increase in plotting over BASIC.

In practice FORTH can run anywhere between 5 to 20 times the speed of BASIC depending on the application and how well written the program is.

Forth supports several other repetitive structures based on the word BEGIN, the first and simplest of these being the BEGIN AGAIN expression.

This structure in essence forms a closed loop within a program and in its simplest form cannot be exited:

```
: CLOSED-LOOP BEGIN ." This will never end " AGAIN ;
```

If you try the above example be warned, you will have to reset the computer and reload FORTH as the program cannot be stopped once it is running. In essence all that happens is that the message which is enclosed between the BEGIN and AGAIN statement is printed on the screen repeatedly. This particular structure is usually used as the main body of a finalised program which usually will need to be of a continuous loop form.

The second BEGIN structure is the BEGIN UNTIL. This structure has a built in test which allows the loop to be exited. The usual format is:-

a) The word BEGIN which marks the beginning of the loop.

b) Any sequence of FORTH words which will end with the data stack holding a boolean flag, either a 1 or 0 representing true or false respectively.

c) The word UNTIL which removes the flag from the stack and if it is true the loop is exited, if however the flag is false the loop returns to the code after BEGIN and is executed once again.

Lets have a look at an example using BEGIN UNTIL.

```
: Y/N ( Accepts yes or no reply. Stack 1 = yes 0 = no )
  BEGIN ." Answer yes or no " KEY
       32 OR DUP 121 =
       IF 1 1 ELSE DUP 110 =
          IF 0 1 ELSE CR ." INCORRECT KEY PRESSED " CR DROP 0
          THEN
       THEN
  UNTIL
  SWAP DROP ;
```

This particular example is a very useful routine which prints a prompt message and waits for a key to be pressed. If the key is either "Y" or "N" (in upper or lower case) then a 1 or 0 is put onto the stack in response which can be used to call an appropriate routine, Lets examine its operation closely.

The routine is called Y/N which helps to describe its function. The parenthisis which follow are equivalent to REM statements in Basic, any text enclosed between them being ignored by the compiler. The BEGIN word indicates the beginning of the loop. The prompt message is first displayed and then the FORTH word KEY waites for a key to be pressed, the ascii equivalent of this character being placed onto the top of the stack. We then convert the ascii number to upper-case (if it is not already so) by logically OR-ing it with decimal 32. The number is then duplicated using DUP and compared to the number 121, which is the ascii number for the letter "Y". IF it is a "Y" the numbers 1 1 are placed on the stack and the loop drops through to the UNTIL which removes the top stack entry and if it is true i.e non zero it exits the loop. The words SWAP DROP exchange the stack contents and remove the initial character which was duplicated at the beginning of the routine. This leaves a 1 on the top of the stack which is to indicate that the response from the keyboard was of a "YES" nature.

If the initial test for a "Y" response fails we then test for a no response by comparing with the number 110 which is the ascii representation for upper-case "N". If this passes the two numbers 0 and 1 are put on the stack. The 1 is to enable UNTIL to exit the loop and the 0 is to indicate the"NO" response from the keyboard. The word CR simply prints a carriage return and line feed thus displaying the text messages on new lines.

If both tests fail then we can assume that an incorrect key was pressed, therefore we print the error message, DROP the duplicated character and place a 0 onto the stack. When UNTIL retrieves this 0 it jumps back to the code after BEGIN which prints the prompt message again and waits for keyboard input.

The last of these types of structures is the BEGIN WHILE REPEAT, whose general format is given below.

a). The word BEGIN which marks the start of the loop.

b). Any sequence of FORTH words which result in a boolean flag being placed on the top of the stack.

c). The word WHILE which tests the stack value and if it is true executes the code following it. (see d). If the stack contents are false the loop is terminated.

d). Any sequence of FORTH words which form the main body of the loop.

e). The word REPEAT which sends control back to the code after BEGIN.

Below is a simple example which should illustrate the function of this structure.

```
: TEST-KEY ( Loops until space-bar is pressed )
      BEGIN ." Press the space-bar " CR
         KEY 32 < >
         WHILE ." I said "
      REPEAT
." Thank you " ;
```

In this example we start by printing the prompt message after the BEGIN. We then perform a carriage return and wait for a key to be pressed, the ascii value of this key being placed onto the stack. We then test to see if the key **was not equal** to 32 (the ascii code for the space-bar). If the result **was not equal** to 32 then the satement was true and a 1 is placed on the stack.

This flag is removed and tested by the WHILE and if true the code between WHILE and repeat is executed and control is sent back to the code after BEGIN i.e. the prompt message. This repeats until we press the space-bar which terminates the loop (the test by WHILE is now false) and prints the "Thank you" message.

Run this program and initially press any key other than the space-bar. Notice how on the second and subsequent passes the messages are printed together as one. This is because there is no carriage return after the message "I said". When you do press the space-bar the loop will fall through and the "Thank you" message will be printed.

4.9 INPUT/OUTPUT

Compared with BASIC, FORTH does not provide easy means for inputing and outputing data to a program. We have already seen three methods. The word KEY which reads a single character from the keyboard and places its ascii value on the top of the stack and the words ." and " which print an enclosed string onto the display. There is also the word . (dot) which prints the top stack entry on the display.

There are several input words in FORTH, however the only one we shall concern ourselves with is the word QUERY. This word reads characters typed in at the keyboard terminated by the enter key, up to a maximum of 128 and stores them in the terminal input buffer which was decribed in section 4.3.

The example below used the word QUERY to create a routine which will read in an integer number from the keyboard and place the number on the stack.

```
: INPUT QUERY 13 WORD HERE NUMBER DROP ;
```

We have called this very useful routine INPUT as it behaves in a similar fashion to BASIC command of the same name.

The word query will allow us to type characters into the keyboard which in this case will be a valid number. The word WORD (no it is not a mistake) reads all text from the terminal input buffer until if reaches a delimiter which was put on the stack before WORD, in this case the 13 which represents a cariage return. WORD transfers the text to the first free location in the dictionary leaving a character count in the first byte. The word HERE puts this dictionary address onto the stack for NUMBER which converts a stream of ascii characters into a valid number, the resulting number being put onto the stack.

So much for inputing information, what about output. Again there are several words provided which allow for outputing data, some of these we have already met. The output operator we will deal with now is the word TYPE. This word takes an address and byte count from the stack and prints out the count of ascii characters at that address.

To illustrate this first enter the screen editor at screen one and type in a line of characters on line 0. Then exit the editor and type in and run the example below.

: PRINT 8 BLOCK 50 TYPE ;

You should see the first 50 characters that were typed in screen 1 displayed on the screen. This routine which we have called PRINT uses a word called BLOCK which puts the address of line 0 screen 1 onto the stack . TYPE takes this and the charactercount of 50 and prints out that ammount of characters to the display.

VARIABLES AND CONSTANTS

4.10

Variables and constants are a concept found in most high level computer languages and FORTH is no exeption. However before we examine these it is a good time to introduce three new FORTH words namely @ , ? and !.

The word @ takes an address from the stack and replaces it with the 16 bit value stored at that address, , (similar to BASICS PEEK command). This value can then be displayed using. (dot). The word ? combines both @ and . (dot) and prints the value directly to the display. For example :-

HEX 3FFB @ or

HEX 3FFB ?

will both print the 16 bit value contained at address 3FFB hex onto the display.

The word ! is roughly equivalent to the BASIC POKE command and takes an address and a 16 bit value from the stack and stores the value at that address e.g.

HEX 13E3 1000 !

will store the value 13E3 into locations 1000 hex and 1001 hex, (remember its a 16 bit value therefore must be stored in two memory locations).

Having examined the above words we can now have a look at variables which are defined in the form:-

(n) VARIABLE name

where (n) is the initial value and name is the label we wish to give the variable e.g.

5 VARIABLE MONTH

will initialise a variable called MONTH to the value 5 (to represent the month of May). We can read this variable using :-

MONTH @ . or more easily
MONTH ?

We can modify the contents of MONTH by using the word ! e.g.

6 MONTH !

A constant is defined in similar fashion e.g.

12 CONSTANT DOZEN

the only difference being that once defined a constant cannot (and should not) be altered. To read the constant we do not need to use the word @ as is required by a variable . we only need say:-

DOZEN .

to display the constant value on the screen.

This then covers our short introduction into the world of FORTH. It is by no means complete nor has it been extensive. In reality we have only scratched the surface of the words available and there use. No attempt has been made to teach FORTH programming as this itself would fill a large volume of text.

What it is hoped is that you have been able to develop a curiosity for the language which you can now develop, if necessary with the help of one of the many texts available on the market. Appendix 5 gives details of books recommended by the author.

APPENDIX 1

ERROR CODES

The FIG-FORTH System issues the following error codes. If the error code is followed by an OK prompt then the message is a warning only. If OK is absent then there was a true error which has caused the system to ABORT its current operation.

ERROR 0 (UNDEFINED)	The word is not in the context or current vocabulary. If VLIST shows it to be present in the dictionary, then the word was never correctly defined, i.e. an error occured during compilation.
ERROR 1 (STACK EMPTY)	An attempt has been made to remove an item from an empty stack.
ERROR 4 (NOT UNIQUE)	The current definition name already exists in the dictionary. The new word will supersede the old one unless it is removed with FORGET.
ERROR 7 (DICTIONARY OR STACK FULL)	This word is issued if the parameter stack or dictionary come within 128 bytes of each other, i.e. close to over-writing one another.
ERROR 17 (COMPILATION ONLY)	The referenced word should only be used during compilation
ERROR 18 (EXECUTION ONLY)	The referenced word should only be used during execution.
ERROR 19 (NOT PAIRED)	The word shown has no pair, e.g. a LOOP without a DO or REPEAT without a BEGIN etc.
ERROR 20 (INCOMPLETE)	The statement has not been completed, e,g, an IF without its completing THEN.
ERROR 21 (PROTECTED)	An attempt has been made to FORGET a word behind the protective FENCE.
ERROR 22 (LOAD ONLY)	The word shown must be used only when loading.

APPENDIX 2

COLD START PARAMETERS

The following is a list of the cold start parameters used at power up. These may be modified by the experienced FORTH user, to adopt the system to their own requirements, by use of the FORTH word +ORIGIN, which places the base address and the offset on the stack.

0C	+ORIGIN	TOPMOST WORD IN FORTH VOCABULARY. This is the address from where dictionary searches will commence. Modified by the FORTH word SYSDUMP.
0E	+ORIGIN	Backspace CHARACTER (included for FIG-FORTH compatability only)
10	+ORIGIN	Pointer to user area.
12	+ORIGIN	Pointer to base of parameter stack.

14	+ORIGIN	Pointer to base of return stack.
16	+ORIGIN	Pointer to base of terminal input buffer.
18	+ORIGIN	Maximum name length. Used by WIDTH and preset to 31.
1C	+ORIGIN	This sets the FENCE below which words will not be forgotten. Points to the last word defined after power up and is modified by the FORTH word SYSDUMP.
1E	+ORIGIN	Address of dictionary pointer at power up. Modified by the FORTH word SYSDUMP.
20	+ORIGIN	Pointer to vocabulary link at power up.

FORTH LANGUAGE GLOSSARY

This glossary contains all of the word definitions utilised in the implementation of FIG-FORTH for the AMSTRAD CPC-464.

The first section details word definitions which are extensions to the basic FIG-FORTH language.
The second section gives details of the FIG FORTH vocabulary.

NOTATION USED

The first line of each definition shows a symbolic view of the stack conditions before and after execution of the word as shown below.

xx xx	— — —	xx xx
Stack condition prior to execution	Execution	Stack condition after execution

In each case, i.e. before and after execution, the top stack parameter is to the right.

The stack notation used is as follows:-

addr	Memory address
b	8 bit byte (MSB of 16 bit word = 0)
c	7 bit ASCII CHARACTER (Top 9 bits = 0)
d	32 bit signed double length integer
f	Boolean flags (either true or false)
ff	Boolean false flag, i.e. 0
n	16 bit signed integer
u	16 bit unsigned integer
tf	Boolean true, i.e. non-zero

APPENDIX 3

FIG-FORTH EXTENSIONS
a). Stack operators:-

.S Non destructive print out of stack contents.

2* n1 n2 --- n3
Multiplies n1 by n2 to leave n3 (faster than 2 *)

2DROP n1 n2 ---
Drops 2 single or 1 double number from the stack.

2DUP n1 n2 --- n1 n2 n1 n2
Duplicates top two numbers or 1 double number

2 OVER n1 n2 n3 n4 --- n1 n2 n3 n4 n1 n2
copies 2nd pair of numbers or 2nd double number to top of stack.

2 SWAP n1 n2 n3 n4 --- n3 n4 n1 n2
Swaps Top two paids of numbers or two double numbers to the stack.

PICK n --- n1
Copies the nth item on the stack to the top.

ROLL n --- n1
Removes the nth item on the stack to the top.

b) SYSTEM OPERATORS

0TIME ---
Sets the internal timer to zero and begins counting cycle

?TIME --- d
Puts the internal timer count onto the stack as a double number. The timer continues counting. The number may be printed using D.

BELL Sounds a short beep

CASE See example
Associated words - OF, ENDOF, ENDCASE

```
: CASE-EXAMPLE (N -- )
CASE
1 OF ."one" ENDOF
2 OF ."two" ENDOF
5 OF ."five" ENDOF
0 OF ."zero" ENDOF
ENDCASE;
```

Case executes the routine associated with the top stack value if present, else control falls through to words after endcase.

CLS --
Clears the V.D.U. screen.

EDITOR The name of the editor vocabulary. Execution makes the editor the context vocabulary.

EDIT n ---
Invokes the screen editor at screen n.

GET n1 n2 ---
Loads screens n1 to n2 from cassette.

MYSELF ---
Compiles C.F.A. of definition being compiled into dictionary therebye allowing a word to "call" itself recursively.

PRINTER ---
Directs all output to the printer

GETKEY --- ff (no key pressed)
--- c tf (Key pressed)
Scans the keyboard for a key depression. If no key is pressed returns a false flag. If a key is pressed returns the ascii code for the key and a true flag.

PUT n1 --- n2
Saves screens n1 to n2 to cassette.

RAND n1 --- n2
Generates a random number n2 in the range 1 to n1

SCREEN —

Directs all output to the V.D.U. used after the PRINTER command.

SYSDUMP —

Saves current version of forth to cassette including all new definitions. Forth may be reloaded in the normal way.

TASK —

A dummy word marking the end of the initial precompiled vocabularies.

TLOAD n1 ADDR —

Loads n1 bytes from cassette and stores at address ADDR.

TSAVE n1 ADDR. —

Saves n1 bytes from addr to cassette

WAIT —

Suspends operation and waits for a key to be pressed. Operation then resumes.

c) GRAPHICS AND COLOURS

BORDER n1 n2 —

Sets Border to a pair of colours n1, n2. For non flashing border n1 = n2

CLG —

Clears the graphics screen.

DRAW n1 n2 —

Draws a line in current graphics pen ink from position of graphics cursor to points n1 and n2

DRAWR n1 n2 ---

Draws a line in current graphics pen ink from the position of the graphics cursor to a point which is the offset of n1 n2.

EXCHANGE n1 n2 ---

Swaps states of streams n1 and n2

GPAPER n1 ---

Sets graphics paper ink to colour n1

GPEN n1 ---

Sets pen number for graphics pen

GWINDOW n1 n2 n3 n4 ---

Creates graphics window n1 = Top, n2 = Bottom n3 = Left, n4 = Right

INK n1 n2 n3 ---

Sets ink n3 to a pair of colours n1 and n2, if non flashing n1 = n2.

INVERSE ---

Exchanges current pen and paper inks

LOCATE n1 n2 ---

Move cursor ot position n1, n2 of current text window. n1 column (1-80) n2 is line (1-25)

MODE n1 ---

selects screen mode, n1 taken mod 4.

MOVE n1 n2 ---

Moves graphics cursor to position n1 (X-DIR) and n2 (Y-DIR)

MOVER n1 n2 ---

Moves the graphics cursor to a point relative of its current position by n1 (X-DIR), n2 (Y-DIR)

ORIGIN n1 n2 ---
Sets the x, y origin of the graphics cursor

PAPER n1 ---
Sets colour of paper to n1

PEN n1 ---
Selects pen for current screen mode

PLOT n1 n2 ---
Plots a point in the currrent graphics pen at location n1, n2

PLOTR. n1 n2 ---
Plots a point on the screen in the current graphics pen at a postion from the current graphics cursor offset by n1 and n2

STREAM n1 ---
Selects text stream n1, n2 must be in the range 1-7

TAG **c1 —**
Sends character c1 to graphics cursor

TEST **n1 n2 —n3**
Test the point on the graphics screen absolute from the cursor, specified by n1 (x-dir) and n2 (y-dir) leaving the current graphics pen number at that point as n3.

TESTR **n1 n2 — n3**
Test the point on the graphics screen offset by n1 and n2 from the graphics cursor. Returns to the top of the stack the graphics pen number n3 at that point.

TRANSPARENT **n1 —**
n1 = 0 Disables transparent option
n1 = 1 Enables transparent option

WINDOW **n1 n2 n3 n4 —**
n1 = Top, n2 = Bottom, n3 = Left, n4 = Right

d) STRINGS AND ARRAYS

1-ARRAY n1 --- (name)
Creates a single dimension array called (name) of n1 elements

2-ARRAY n1 n2 --- (name)
Creates a two dimension array called (name) of n1 rows by n2 columns.

STRING n1 --- (name)
Creates string storage space of n1 spaces called (name)

INPUT$ pfa n ---
Used to read input characters into a predifined storage space (see STRING).

e) SOUND FACILITIES

AMP-ENV n1 (name) ---
Assigns amplitude envelope n to sound block (name)

CHANNEL n1 (name) ---
Assigns channel(s) n1 to sound block (name)

CONTINUE ---
Releases all sounds which have been held

DURATION n1 (name) ---
Sets the duration of sound block (name) to length n1

ENT
Creates a tone envelope. For a full description refer to section 3.12

ENV

Creates an amplitude envelope. For a full description refer to section 3.12

FREEZE ---

Stops all sounds in mid flight. may be restarted by using CONTINUE.

NOISE n1 (name) ---

Adds a noise component n1 to sound block (name).

PERIOD n1 (name) ---

Sets the period of the note of sound block (name) to n1.

PLAY (name) ---

Executes the sound block (name). if the sound queues are full returns an error message and executes ABORT.

RELEASE n1 ---

Releases a sound on channel(s) n1 which was previously held using SHOLD.

RENDEZVOUS n1 (name) ---

Will cause sound block (name) to rendezvous with channel(s) (n1).

RESET ---

Reset the sound manager and shuts down the sound chip causing all sounds to stop immidiately.

SFLUSH (name) ---

Sets the flush bit in sound block (name)

SHOLD (name) ---

Sets the hold bit of sound block (name)

SOUND

This is the main sound block creation definition. For a full description refer to section 3.12

SQ (n) ---

Returns the status of channel (n)

SWAIT (n) ---

Forces channel (n) to wait until there is space on the sound queue before issuing any more sounds to the queue.

TONE-ENV n1 (name) ---

Assigns tone envelopes n1 to sound block (name)

VOLUME n1 (name) ---

Sets the volume of sound block (name) to n1

! n addr ---
Store 16 bits of n at address. Pronounced "store".

!CSP
Save the stack position in CSP. Used as part of the compiler security.

d1 --- d2
Generate from a double number d1, the next ascii character which is placed in an output string. Result d2 is the quotient after division by BASE, and is maintained for further processing. Used between <# and #>. See #S

#> d --- addr count
Terminates numeric output conversion by dropping d, leaving the text address and character count suitable for TYPE.

#S d1 --- d2
Generates ascii text in the text output buffer, by the use of #, until a zero double number n2 results. Used between <#and#>

' --- addr
Used in the form:' nnnn. As a compiler directive, executes in a colon-definition to compile the address as a literal. If the word is not found after a search of CONTEXT and CURRENT, an appropriate error message is given. Pronounced "tick".

(
Used in the form: (cccc). Ignore a comment that will be delimited by a right parenthesis on the same line. May occur during execution or in a colon-definition. A blank after the leading parenthesis is required.

(.")
The run-time procedure, compiled by ." which transmits the following in-line text to the selected output device. See ."

(;CODE)
The run-time procedure, compiled by ;CODE, that rewrites the code field of the most recently defined word to point to the following machine code sequence. See ;CODE.

(+LOOP) n ---
The run-time procedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP.

(ABORT)
Executes after an error when WARNING is -1. This word normally executes ABORT, but may be altered (with care) to a user's alternative procedure.

(DO)
The run-time procedure compiled by DO which moves the loop control parameters to the return stack. See DO.

(FIND) addr1 addr2 --- pfa b tf (ok)
addr1 addr2 --- ff (bad)
Searches the dictionary starting at the name field address addr2, matching to the text at addr1. Returns parameter field address, length byte of name field and boolean true for a good match. If no match is found, only a boolean false is left.

(LINE) n1 n2 --- addr count
Convert the line number n1 and the screen n2 to the disc buffer address containing the data. A count of 64 indicates the full line text length.

(LOOP)

The run-time **procedure** compiled by LOOP which increments the loop index and test for loop completion. See LOOP.

(NUMBER) dl addr --- d2 addr2

Convert the ascii text beginning at addr1+1 with regard to BASE. The new value is accumulated into double number dl, being left as d2. Addr2 is the address of the first unconvertable digit. Used by NUMBER.

* nl n2 --- prod

Leave the signed product of two signed numbers.

*/ nl n2 n3 --- n4

Leave the ratio n4 - nl*n2/n3 where all are signed numbers. Retention of an intermediate 31 bit product permits greater accuracy than would be available with the sequence: nl n2 * n3 /

*/MOD nl n2 n3 --- n4 n5

Leave the quotient n5 and remainder n4 of the operation nl*n2/n3 A 31 bit bit intermediate product is used as for */.

\+ nl n2 --- sum

Leave the sum of nl+n2.

+! nl addr ---

Add n to the value at the address. Prounced "plus-store"

+- nl n2 --- n3

Apply the sign of n2 to nl, which is left as n3

+BUF addl --- addr2 f

Advance the disc buffer address addrl to the address of the next buffer addr2. Boolean f is false when addr2 is the buffer presently pointed to by variable PREV.

+LOOP nl --- (run)

addr n2 --- (complete)

Used in a colon definition in the form: DO ... nl +LOOP. At run-time, +LOOP selectively controls branching back to the corresponding DO based on nl, the loop index and the loop limit. The signed increment nl is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (nl > 0), or until the new index is equal to or less than the limit (nl < 0). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, +LOOP compiles the run-time word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO. n2 is used for compile time error checking.

+ORIGIN n --- addr

Leave the memory address relative by n to the origin parameter area. n is the minimum address unit, either byte or word. This defintion is used to access or modify the boot-up parameters at the origin area.

, n ---

Store n into the next available dictionary memeory cell, advancing the dictionary pointer. (comma)

\- nl n2 --- diff

Leave the difference of nl-n2.

-->

Continue interpretation with the next disc screen. (pronounced next-screen).

-DUP n1 -- n1 (if zero)
-- n1 n1 (non-zero)
Reproduce n1 only if it is non-zero. This is usually used to copy a value just before IF, to eliminate the need for an ELSE part to drop if.

-FIND --- pfa b tf (found)
--- ff (not found)
Accepts the next text word (delimited by blanks) in the input stream to HERE, and searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean false is left.

-TRAILING addr n1 --- addr n2
Adjusts the character count n1 of a text string beginning at address to suppress the output of trailing blanks. i.e. the characters at addr+n1 are blanks.

n ---
Print a number from signed 16 bit two's complement value, converted according to other nemeric BASE. A trailing blanks follows. Pronounced "dot".

." Used in the form:
." cccc"
Compiles an in-line string cccc (delimited by the trailing ") with an execution proceedure to transmit the text to the selected output device. If executed outside a definition, ." will immediately print the text until the final ". The maximum number of characters may be an installation dependent value. See (.").

.LINE line scr ---
Print on the terminal device, a line of text form the disc by its line and screen number. Trailing blanks are suppressed.

.R n1 n2 ---
Print the number n1 right aligned in a field whose width is n2. No following blank is printed.

/ n1 n2 --- quot
Leave the signed quotient of n1/n2.

/MOD n1 n2 --- ram quot
leave the remainder and signed quotient of n1/n2. The remainder has the sign of the dividend.

0 1 2 3 --- n
These small numbers are used so often that it is attractive to define them by name in the dictionary as constants.

0< n --- f
Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0= n ---f
Leave a true flag if the number is equal to zero, otherwise leave a false flag.

0BRANCH f --
The run-time proceedure to conditionally branch. if f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF, UNTIL, and WHILE.

1+ n1 --- n2
Increment n1 by 1.

2+ n1 --- n2
Leave n1 incremented by 2.

Used in the form called a colon definition

:cccc ...;

Creates a dictionary entry defining cccc as equivalent to the following sequence of Forth word definitions '—' until the next ';' or ';CODE'. The compiling process is done by the text interpreter as long as STATE is non-zero. Other detail are that the CONTEXT vocabulary is set to the CURRENT vocabulary and that words with the precedence bit set (P) are executd rather than being compiled.

;

Terminate a colon-definition and stop further compilation. Compiles the run-time ;S.

;CODE

used in the form : cccc .... ;CODE assembly mnemonics. Stop compilation and terminate a new defining word cccc by compiling (;CODE). Set the CONTEXT vocabulary to ASSEMBLER, assembling to machine code the following mnemonics.
When cccc later executes in the form: cccc nnnn. The word nnnn will be created with its execution proceedure given by the machine code following cccc. That is, when nnnn is executed, it does so by jumping to the code after nnnn. An existing defining word must exist in cccc prior to ;CODE.

;S

Stop interpretation of a screen. ;S is also the run-time word compiled at the end of a colon-definition which returns execution to the calling proceedure.

< n1 n2 — f

Leave a true flag if n1 is less than n2; otherwise leave a false flag.

<#

Setup for pictured numeric output formatting using the words:
#S SIGN # .
The conversion is done on a double number producing text at PAD.

<BUILDS

Used within a colon-definition: :cccc <BUILDS ... DOES> ... ;
Each time cccc is executed, <BUILDS defines a new word with a high-level execution procedure. Executing cccc in the form:

cccc nnnn

uses BUILDS to create a dictionary entry for nnnn with a call to the DOES> part for nnnn. When nnnn is later executed, it has the address of its parameter area on the stack and executes the words after DOES> in cccc. <BUILDS and DOES> allow run time procedure to written in high level rather than in assembler code (as required by ;CODE).

= n1 n2 --- f

Leave a true flag if n1 = n2; otherwise leave a false flag.

> n1 n2 --- f

Leave a true flag if n1 is greater than n2; otherwise a false flag.

>R n ---

Remove a number form the computation stack and place as the most accessable on the return stack. Use should be balanced with >R in the name definition.

? addr --

Print the value contained at the address in free format according to the current base.

?COMP
Issue error message if not compiling.

?CSP
Issue error message if stack position differs from value saved CSP.

?ERROR f n ---
Issue an error message number n, if the boolean flag is true.

?EXEC
Issue an error message if not executing

?LOADING
Issue an error message if not loading

?PAIRS n1 n2 ---
Issue an error message if n1 does not equal n2. The message indicates that compiled conditions do not match.

?STACK
Issue an error message if the stack is out of bounds. This definition may be installation dependent.

?TERMINAL --- f
Perform a tst of the terminal key-board for actuation of the CTRL/Q key. A true flag indicates actuation.

@ addr --- n
Leave the 16 bit contents of address.

ABORT
Clear the stacks and enter the execution state. Return control to the operators terminal, printing a message appropriate to the installation.

ABS n ---u
Leave the absolute value of n as u.

AGAIN addr n --- (compiling)
Used in a colon-definition in the form:
BEGIN ... AGAIN
At run-time, AGAIN forces execution to return to corresponding BEGIN. There is no effect on the stack. Execution cannot leave this loop (unless R DROP is executed one level below).
At compile time, AGAIN compiles BRANCH with an offset from HERE to addr. n is used for compile-time error checking.

ALLOT n ---
Add the signed number to the dictionary pointer DP. May be used to reserve dictionary space or re-origin memory. n is with regard to computer address type (byte or word).

AND n1 n2 --- n2
Leave the bitwise logical and of n1 and n2 as n3.

B/BUF --- n
This constant leaves the number of bytes per disc buffer, the byte count read from disc by BLOCK.

B/SCR --- n
This constant leaves the number of blocks per editing screen. By convention, an editing screen is 1024 bytes organized as 16 lines of 64 characters each.

BACK addr ---
Calculate the backward branch offset from HERE to addr and compile into the next available dicitionary memory address.

BASE --- addr
A user variable containing the currrent number base used for input and output conversion

BEGIN --- addr n (compiling)
Occurs in a colon-definition in form:
BEGIN ... UNTIL
BEGIN ... AGAIN
BEGIN ... WHILE ... REPEAT
At run-time, BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL, AGAIN or REPEAT. When executing UNTIL, a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs.
At compile time BEGIN leaves its return address and n for compiler error checking.

BL --- c
A constant that leaves the ascii value for "blank".

BLANKS addr count ---
Fill an area of memory beginning at addr with blanks.

BLK --- addr
A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.

BLOCK a --- addr
Leave the memory address of the block buffer containing block n. If the block is not already in memory, it is transferred from disc to which ever buffer was least recently written. If the block occupying that buffer has been marked as updated, it is rewritten to disc before block n is read into the buffer. See also BUFFER, R/W UPDATE FLUSH.

BRANCH
The run-time procedure to unconditionally branch. An in-line offset is added to the inerpretive pointer IF to branch ahead or back. BRANCH is compiled by ELSE, AGAIN, REPEAT

BUFFER n --- addr
Obtain the next memory buffer, assigning it to block n. If the coments of the buffer is marked as updated, it is written to the disc. The address left is the first call within the buffer for data storage.

C! b addr ---
Store 8 bits at address. On word addressing computers, furtherspecification is necessary regarding byte addressing.

C. b ---
Store 8 bits of b into the next available dictionary byte, advancing the dictionary pointer. This is only available on byte addressing computers, and should be used with caution on byte addressing mini-computers.

C@ addr --- b
Leave the 8 bit contents of memory address. On word addressing computers, further specification is needed regarding byte addressing.

CFA pfa ---cfa

Convert the parametrer field address of a definition to its code field address

CMOVE from to count ---

Move the specified quantity of bytes beginning at address from to address to. The contents of address from is moved first proceeding toward high memory. Further specification is necessary on word addressing computers.

COLD

The cold start proceedure to adjust the dictionary pointer to the minimum standard and restart via ABORT. May be called from the terminal to remove application programs and restart.

COMPILE

When the word containing COMPILE executes, the execution address of the word following COMPILE is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in addition to simply compiling an execution address (which the interpreter already does).

CONSTANT n ---

A defining word used in the form:

n CONSTANT cccc

to create word cccc, with its parameter field containing n. When cccc is later executed, it will push the value of n to the stack.

CONTEXT --- addr

A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.

COUNT addr1 ---addr2 n

Leave the byte address addr2 and byte count n of a message text beginning at address addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text starts with the second byte. Typically COUNT is followed by TYPE.

CR

Transmit a carriage return and line feed to the selected output device.

CREATE

A defining word used in the form:

CREATE cccc

by such words as CODE and CONSTANT to create a dictionary header for a Forth definition. The code field contains the address of the words parameter field. The new word is created in the CURRENT vocabulary

CSP --- addr u

A user variable temporarily storing the stack pointer position, for compilation error checking.

D+ d1 d2 ---dsum

Leave the double number sum of two double numbers.

D+- d1 n --- d2

Apply the sign of n to the double number d1, leaving it as d2.

D. d ---

Print a signed double number from a 32 bit two's a complement value. The high-order 16 bits are most accessable on the stack. Conversion is performed according to the current BASE. A blank follows. Pronounced D-dot.

D.R d n ---
Print a signed double number d right aligned in a field n characters wide.

DABS d ---ud
Leave the absolute value ud of a double number.

DECIMAL
Set the numeric conversion BASE for decimal input-output.

DEFINITIONS
Used in the form:
cccc DEFINITIONS
Set the CURRENT vocabulary to the CONTEXT vocabulary. In the example, executing vocabulary name cccc made it the CONTEXT vocabulary and executing DEFINITIONS made both specify vocabulary cccc.

DIGIT c nl --- n2 tf (ok)
c nl --- ff (bad
converts the ascii character c (using base nl) to its binary equivalent n2, accompanied by a true flag. Ifthe conversion is invalid, leaves only a false flag.

DLITERAL d --- d (executing)
d --- (compiling)
If compiling, compile a stack double number into a literal will push it to the stack. If executing, the number will remain on the stack.

DMINUS dl ---d2
Convert dl to its double number two's complement.

DO nl n2 --- (execute)
addr n ---(compile)
Occurs in a colon-definition in form:
DO ... LOOP
DO ... +LOOP
At run time, DO begins a sequence with repetitive execution controlled by a loop limit nl and an index with initial value n2. DO removes these the index is incremented by one. Until the new index equals or execeeds the limit, execution loops back to just after DO; otherwise the loop parameters are discarded and execution continues ahead. Both nl and n2 are determined at run-time and may by the result of the other operations. Within a loop I will copy th current value of the index to the stack. See I, LOOP, +LOOP, LEAVE. When compiling within the colon-definition, DO compiles (DO), leaves the following address addr and n for later error checking.

DOES>
A word which defines the run-time action within a high-level defining word. DOES > alters the code alters the code field and first parameter of the new word to execute the sequence of compiled word addresses following DOES>. Used in combination with < BUILDS. When the DOES > part executes it begins with the addr of the first parameter of the new word on the stack. This allows inerpretation using this area or its contents. Typical uses include the Forth assembler, multidiminsional arrays, and compiler generation.

DP --- addr
A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by HERE and altered by ALLOT.

DPL --- addr
A user variable containing the number of digits to the right of the decimal on double integer input. It may also be used hold output column location of a decimal point, in user generated formating. The default value on single number input is -1.

DROP n ---
Drop the number from the stack.

DUMP addr n ---
Print the contents of n memory locations beginning at addr. Both addresses and contents are shown in the current numeric base.

DUP n --- n n
Duplicate the value on the stack.

ELSE addr1 n1 --- addr2 n2
(compiling)
Occurs within a colon-definition in the form:
IF ... ELSE ... ENDIF
At run-time, ELSE executes after the true part following IF. ELSE forces execution to skip over the following false part and resumes execution after the ENDIF. If has no stack effect.
At compile-time ELSE emplaces BRANCH reserving a branch OFFSET, leaves the address addr2 and n2 for error testing. ELSE also resolves the pending forward branch form IF by calculating the offset from addr1 to HERE and storing at addr1.

EMIT c ---
Transmit asciii character c to the selected output device. OUT is incremented for each character output.

EMPTY-BUFFERS
Mark all block-buffers as empty, not necesssarily affecting the contents., Updated blocks are not written to the disc. This is also an initialisation proceedure before first use of the disc.

ENCLOSE addr1 c ---
addr1 n1 n2 n3
The text scanning primitive used by WORD. From the text address addr1 and an ascii delimiter the byte offset to the first non-delimiter character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included. This proceedure will not process past an ascii 'null', treating it as an unconditional delimiter.

END
This is an alias or duplicate definition for UNTIL.

ENDIF addr n --- (compile)
Occurs in a colon-definition in form:
IF ... ENDIF
IF ... ELSE ... ENDIF
At run-time, ENDIF serves only as thje destination of a forward branch from IF or ELSE. It marks the conclusion of the condional structure. THEN is another name for ENDIF. Both names are supported in fig-FORTH. See also IF and ELSE. At compile-time, ENDIF computers the forward branch offset from addr to HERE and stores it at addr. n is used for error tests.

ERASE addr n ---
Clear a region of memory to zero from addr over n addresses.

ERROR line --- in blk
Execute error notification and restart of system. WARNING is first examined. If 1, the text of line n, relative to screen 4 to drive 0 is printed. This line number may be positive or negative and beyond just screen 4. If WARNING is 0, n is just printed as a message number (non disc installation). If WARNING is -1 the definition (ABORT) is executed which executes the system ABORT. The user may cautiously modify this execution by alterin (ABORT), fig-FORTH saves the contents of IN and BLK to assist in determining the location of the error. Final action is execution of QUIT.

EXECUTE addr --
Execute the defintion whose code field address is on the stack. The code field address is also called the compilation address.

EXPECT addr count ---
Transfer characters from the terminal to address, until a "return" or the count of characters have been received. One or more nulls are added at the end of the text.

FENCE --- addr
A user variable containing an address below which FORGETTING is trapped., To forget below this point the user must alter the contents of FENCE.

FILL addr quan b ---
Fill memory at the address with the specified quantity of bytes b.

FIRST --- n
A constant that leaves the address of the first (lowest) block buffer.

FORGET
Executd in the form:
FORGET cccc
Deletes defintion named cccc from the dicionary with all entries physically following it. In fig-FORTH, an error message will occur if the CURRENT and CONTEXT vocabularies are not currently the same.

FORTH
The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. Until additional user vocabularies are defined, new definitions become a part of FORTH. FORTH is immediate, so it will execute during the creation of a colon definition, to select this vocabulary at compile time.

HERE --- addr
Leave the address of the next available dictionary location.

HEX
Set the numeric conversion base to sixteen (hexadecimal).

HLD ---addr
A user variable that holds the address of the latest character of text during numeric output convesion.

HOLD C ---
Used between < # and #> to insert an ascii character into a pictured numeric output string. e.g. 2E HOLD will place a decimal point.

I --- n
Used within a DO-LOOP to copy the loop index to the stack. Other use is implementation dependent. See R.

ID. addr ---
Print a definition's name from its name field address.

IF f --- (run time)
--- addr n (compile)
Occurs is a colon-defintion in form:
IF (tp) ... ENDIF
IF (tp) ... ELSE (fp) ... ENDIF
At run-tiem, IF selects execution based on a boolean flag. If f is true (non-zero), execution continues ahead thru the true part. If f is false (zero), execution skips till just after ELSE to execute the false part. After either part, execution resumes after ENDIF. ELSE and its false part are optional.; if missing, false execution skips to just after ENDIF.

ENDIF
At compile-time IF compiles 0BRANCH and reserves space for an offset at addr. addr and n are used later for resolution of the offset an error testing.

IMMEDIATE
Mark the most resently made definition so that when encountered at compile time, it will be executed rather than being compiled. i.e. the precedence bit in its header is set. This method allows definitions to handle unusual compiling situations, rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceeding it with [COMPILE].

IN --- addr
A user variable containing the byte offset within the current input text buffer (terminal or disc) from which the next text will be accepted. WORD uses and moves the value of IN.

INDEX from to ---
Print the first line of each screen over the range from, to. This is used to view the comment lines of an area of text on disc screens.

INTERPRET
The outer text interpreter which sequentialy executes or compiles text from the input stream (terminal or disc) depending on STATE. If the word name cannot be found after a search of CONTEXT and then CURRENT it is converted to a number according to the current base. That also failing, an error message echoing the name with "?" will be given. Text input will be taken according to the convention for WORD. if a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than to force this action. See NUMBER.

KEY ---c
Leave the ascii value of the next terminal key struck.

LATEST --- addr
Leave the name field address of the topmost word in the CURRENT vocabulary.

LEAVE
Force termination of a DO-LOOP at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until LOOP or +LOOP is encountered.

LFA pfa --- lfa
Convert the parameter field address of a dictionary definition to its link field address.

LIMIT --- n

A constant leaving the address just above the highest memory available for a disc buffer. Usually this is the highest system memory.

LIST n ---

Display the ascii text of screen n on the selected output device. SCR contains the screen number during and after this process.

LIT --- a

Within a colon-definition, LIT is automatically compiled before each 16 bit literal number encountered is input text. Later execution of LIT causes the contents of the next dictionary address to be pushed to the stack.

LITERAL n --- (compiling)

If compiling, then compile the stack value n as a 16 bit literal. This definition is immediate so that it will execute during a colon definition. The intended use is:

: xxx (calculates) LITERAL ;

Compilation is suspended for the compile time calculation of a value. Compilation is reusumed and LITERAL compiles this value.

LOAD n ---

Begin interpretation of screen n. Loading will terminate at the end of the screen or at ;S. See ;S and -->

LOOP addr n --- (compiling)

Occurs in a colon-definition in form:

DO ... LOOP

At run-time, LOOP selectively controls branching back to the corresponding DO based on the loop index and limit. The loop index is incremented branch back to DO occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and execution continues ahead. At compile-time, LOOP compiles (LOOP) and uses addr to calculate an offset to DO. n is used for error testing.

M* n1 n2 --- d

A mixed magnitude math operation which leaves the double number signed product of two signed numbers.

M/ d n1 --- n2 n3

A mixed magnitude math operator which leaves the signed remainder n2 and signed quotient n3, from a double number dividend and divisor n1. The remainder takes its sign from the dividend.

M/MOD ud1 u2 --- u3 ud4

An unsigned mixed magnitude math operator which leaves a double quotient ud4 and remainder u3, form a double dividend ud1 and single divisor u2.

MAX n1 n2 --- max

Leave the greater of two numbers.

MESSAGE n ---

Print on the selected output device the text of line n relative to screen 4 of drive 0. n may be positive or negative. MESSAGE may be used to print incidental text such as report headers. If WARNING is zero, the message will simply be printed as a number (disc un-available).

MIN n1 n2 --- min

Leave the smaller of two numbers.

MINUS n1 n2 ---

Leave the two's complement of a number.

MOD n1 n2 --- mod
Leave the remainder of n1/n2, with the same sign as n1.

NEXT
This is the inner interpreter that uses the interpretive pointer IP to execute compiled Forth definitions. It is not directly executed but is the return point for all code procedures. It acts by fetching the address pointed to by the address pointed to by W. W points to the code field of a definition which contains the address of the code which executes for that definition. This usage of indirect threaded code is a major contributor to the power, portability, and extensibility of Forth. Location of IP and W are computer specific.

NFA pfa --- nfa
Convert the parameter field address of a definition to its name field.

NUMBER addr --- d
Convert a character string left at addr with a preceeding count, to a signed double number, using the current numeric base. If a decimal point is encountered in the text, its position will be given in DPL, but no other effect occurs. If numeric conversion is not possible, an error message will be given.

OFFSET --- addr
A user variable which may contain a block offset to disc drives. The contents of OFFSET is added to the stack number by BLOCK. Messages by MESSAGE are independent of OFFSET. See BLOCK, DRO, DR1, MESSAGE

OR n1 n2 --- or
Leave the bit-wise logical or of two 16 bit values.

OUT --- addr
A user variable that contains a value incremented by EMIT. The user may alter and examine OUT to control display formating.

OVER n1 n2 --- n1 n2 n1
Copy the second stack value, placing it as the new top.

PAD --- addr
Leave the address of the text output buffer, which is a fixed offset above HERE.

PFA nfa --- pfa
Convert the name field address of a compiled definition to its parameter field address.

PREV --- addr
A variable containing the address of the disc buffer most recently referanced. The UPDATE command marks this buffer to be later written to disc.

QUERY
Input 80 characters of text (or until a "return")from the operators terminal. Text is positioned at address contained in TIB with IN set to zero.

QUIT
Clear the return stack, stop compilation, and return control to the operators terminal. No message is given.

R --- n
Copy the top of the return stack to the computation stack.

R# --- addr U
A user variable which may contain the location of an editing cursor, or other file related function.

R/W · addr blk f —

The fig-FORTH standard disc read-write linkage, addr specifies the source or destination block buffer. blk is the sequential number of the referenced block; and f is a flag for f-o write and f-1 read. R/W determines the location on mass storage, performs the read-write and performs any error checking.

R > — n

Remove the top value from the return stack and leave it on the computation stack. See >R and R.

R0 — addr

A user variable containing the initial location of the return stack. Pronounced R-zero. See RP!

REPEAT addr n — (compiling)

Used within a colon-definition in the form:
BEGIN ... WHILE ... REPEAT
At run-time, REPEAT forces an unconditional branch back to just after the corresponding BEGIN.
At compile-time, REPEAT compiles BRANCH and the offset from HERE to addr. n is used for error testing.

ROT n1 n2 n3 — n2 n3 n1

Rotate the top three values on the stack, bringing the third to the top.

RP!

A computer depenent proceedure to initialize the return stack pointer from user variable R0.

S-> D n — d

Sign extend a single number to form a double number.

S0 — addr

A user variable that contains the initial value for the stack pointer. Pronounced S-zero. See SP!

SCR — addr

A user variable containing the screen number most recently reference by LIST.

SIGN n d — d

Stores an ascii "-" sign just before a converted numeric output string in the text output buffer when n is negative. n is discarded, but double number d is maintained. Must by used between < # and # > .

SMUDGE

Used during word definition to toggle the "smudge bit" in a definitions name field. This prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error.

SP!

A computer dependent proceedure to initialize the stack pointer from S0 .

SP@ — addr

A computer dependent proceedure to return the address of the stack position to the top of the stack, as it was before SP @ was executed.
(e.g. 1 2 SP@ @ . . . would type 2 2 1)

SPACE

Transmit an ascii blank to the output device.

SPACES n —

Transmit n ascii blanks to the output device.

STATE — addr
A user variable containing the compilation state. A non-zero value indicates compilation. The value itself may be implementation dependent.

SWAP n1 n2 — n2 n1
Exchange the top two values on the stack.

THEN
An alias for ENDIF.

TIB — addr
A user variable containing the address of the terminal input buffer.

TOGGLE addr b —
Complement the contents of addr by the bit pattern b.

TRAVERSE addr1 n — addr2
Move across the name field of a fig-FORTH variable length name field. addr1 is the address of either the length byte or the last letter. If n=-1, the motion is toward low memory. The addr2 resulting is address of the other end of the name.

TRIAD scr —
Display on the selected ouput device the three screens which include that numbered scr, beginning with a screen evenly divisible by three. Output is suitable for source text records, and includes a reference line at the bottom taken from line 15 of screen4.

TYPE addr count —
Transmit count charcters from addr to the selected output device.

U* u1 u2 — ud
Leave the unsigned double number product of two unsigned numbers.

U/ ud u1 — u2 u3
Leave the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1.

UNTIL f — (run-time)
addr n — (compile)
Occurs within a colon-definition in the form:
BEGIN ... UNTIL
At run-time, UNTIL controls the conditional brach back to the corresponding BEGIN. If f is false, execution returns to just after BEGIN; if true, exection continues ahead. At compile-time, UNTIL compiles (0BRANCH) and an offset from HERE to addr. n is used for error tests.

UPDATE
Marks the most recently referenced block (pointed to by PREV) as altered. The block will subsequently be transferred automatically to disc should its buffer be required for storage of a different block.

USE — addr
A variable containing the address of the block buffer to use next, as the least recently written.

USER n —
A defining word used in the form:
n USER cccc
Which creates a user variable cccc. The parameter field of cccc contains n as a fixed offset relative to the user pointer register UP for this user variable. When cccc is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable.

VARIABLE

A defining word used in the form:
n VARIABLE cccc
When VARIABLE is executed, it creates the definition cccc with its parameter field initialized to n. When cccc is later executed, the address of its parameter field (containing n) is left on the stack, so that a fetch or store may access this location.

VOC-LINK --- addr

User variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for FORGETting multiple vocabularys.

VOCABULARY

A defining word used in the form:
VOCABULARY cccc
to create a vocabulary definition cccc. Subsequent use of cccc will make it the CONTEXT vocabulary which is searched first by INTERPRET. The sequence "cccc DEFINITIONS" will also make cccc the CURRENT vocabulary into which new definitions are placed. In fig-FORTH, cccc will be so chained as to include all definitions of the vocabulary in which cccc is itself defined. All vocabularys ultimately chain to forth. By convention, vocabualry names are to be declared IMMEDIATE. See VOC-LINK.

VLIST

List the names of the definitions in the context vocabulary. "Break" will terminate the listing.

WARNING --- addr

A user variable containing a value controlling messages. If=1 disc is present, and screen 4 of drive 0 is the base location for messages. If = 0, no disc is present and messages will be presented by number. If = -1, execute (ABORT) for a user specified procedure. See MESSAGE, ERROR.

WHILE f --- (run-time)
adl n1 --- ad1 n1 ad2 n2

Occurs in a colon-definition in the form:
BEGIN ... WHILE (tp) ... REPEAT
At run-time, WHILE selects conditional execution base on boolean flag f. If f is true (non-zero), WHILE continues execution of the true part thru to REPEAT, which then branches back to BEGIN. If f is false (zero), execution skips to just after REPEAT, exiting the structure. At compile time. WHILE emplaces (OBRANCH) and leaves ad2 of the reserved offset. The stack values will be resolved by REPEAT.

WIDTH --- addr

In fig-FORTH, a user variable containing the maximum number of letters saved in the compilation of a definitions name. It must be 1 thru 31, with a default value of 31. The name character count and its natural characters are saved, up to the value in WIDTH. The value may be changed at any time within the above limits.

WORD C ---

Road the next text characters from the input stream being interpreted, until a delimiter c is found, storing the packed character string beginning at the dictionary buffer HERE. WORD leaves the character count in the first byte the cnaracters and ends with two or more blanks. Leading occurances of c are ignored. If BLK is zero, text is taken from the terminal unput buffer, otherwise from the disc block stored in BLK. See BLK, IN.

XOR n1 n2 --- xor

Leave the bitwise logical exclusive or of two values.

[

Used in a colon-definition in form:

: xxx [words [more :

Suspend compilation. The words after [are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with]. See LITERAL,].

[COMPILE]

: xxx [COMPILE] FORTH ;

[COMPILE] will force the compilation of an immediate definition that would otherwise execute during compilation. The above example will select the FORTH vocabulary when xxx executes, rather than at compile time.

]

Resumes compilation, to the completion of a colon-definition. See [.

APPENDIX 5

FURTHER READING.

The titles given here have all been consulted by the author in the writing of this version of FORTH and as such all can be recommended.

1). THREADED INERPRETIVE LANGUAGES by R.G. Loeliger.
Byte Publicaitons

This book presents a revealing insight into the world of FORTH like langauges. It is of particular interest to those with experience of programming at the machine language level. An ideal book for the "hacker" who likes to know what makes things tick.

2). STARTING FORTH By Leo Brodie.
Prentice Hall

This book was written by a member of FORTH inc. which is the company founded by FORTH's inventor Charles Moore. It is an extremely good book with a pleasant and often ammusing style of writing.

3). FORTH FOR MICROS By Steve Oakey.
Newnes Programming Books.

A very good book which gives many examples of FORTH together with there BASIC and PASCAL equivalents. Each chapter ends with self test exercise with sample answers provided.

4). BYTE Magazine, volume 5, number 8, August 1980

An entire issue of this magazine devoted to the FORTH language. This issue is now out of print but certainly worth reading if you can lay your hands on a copy. Regarded by some people as a FORTH bible.

APPENDIX 6

FIG-FORTH VOCABULARY

TASK WHERE EDIT EDITOR PLAY SWAIT ENT ENV (TENV) (AENV) RELEASE SQ (SOUND-QUEUE) FREEZE CONTINUE RESET SFLUSH SHOLD RENDEZVOUS DURATION VOLUME NOISE PERIOD TONE-ENV AMP-ENV CHANNEL SOUND INPUT$ STRING 2-ARRAY 1-ARRAY BELL COPY GETKEY PRINTER (PRINTER) SCREEN TESTR TEST GPAPER GWINDOW GPEN DRAWR DRAW PLOTR PLOT MOVER MOVE ORIGIN CLG EXCHANGE STREAM TAG CHARACTER SYMBOL BORDER PAPER PEN INK MODE INVERSE TRANSPARENT WINDOW JOY1 JOY0 LOCATE ?TIME 0TIME WAIT DUMP CLS ENDCASE ENDOF OF CASE RANDOM NCASE SEED 2OVER 2SWAP 2DROP .S ROLL PICK OCTAL BINARY 2/ 2- 1- 2* <> <= >= NOT MYSELF SYSDUMP SYSWRITE GET PUT TLOAD TSAVE FORTH VLIST TRAID INDEX U. ? . D. .R D.R #S # SIGN #> <# SPACES WHILE ELSE IF REPEAT AGAIN END UNTIL +LOOP LOOP DO THEN ENDIF BEGIN BACK FORGET MESSAGE LIST --> LOAD FLUSH R/W BLOCK BUFFER .LINE (LINE) EMPTY-BUFFERS UPDATE +BUF #BUF PREV USE ' M/MOD */ */MOD MOD / /MOD * M/ M* MAX DABS ABS D+- +- S->D U< MIN COLD WARM ABORT (DEFINITIONS VOCABULARY IMMEDIATE INTERPRET ?STACK DLITERAL LITERAL [COMPILE] CREATE ID. ERROR (ABORT) -FIND NUMBER (NUMBER) WORD PAD HOLD BLANKS ERASE FILL QUERY EXPECT ." (.") -TRAILING TYPE COUNT DOES> <BUILDS ;CODE (;CODE) DECIMAL HEX SMUDGE] [COMPILE ?LOADING ?CSP ?PAIRS ?EXEC ?COMP ?ERROR !CSP PFA NFA CFA LFA LATEST TRAVERSE -DUP SPACE ROT > < = C, , ALLOT HERE 2+ 1+ HLD R# CSP FLD DPL BASE STATE CURRENT CONTEXT OFFSET SCR OUT IN BLK VOC-LINK DP FENCE WARNING WIDTH TIB R0 S0 +ORIGIN QUIT B/SCR B/BUF LIMIT FIRST C/L BL 3 2 1 0 USER VARIABLE CONSTANT ; : DMINUS MINUS- D+ + 0< 0= R R> >R LEAVE ;S RP! RP@ SP! SP@ XOR OR AND U/ U* CMOVE ENCLOSE (FIND) DIGIT K J I (DO) (+LOOP) (LOOP) BRANCH 0BRANCH CR ?TERMINAL KEY (KEY) EMIT TOGGLE ! C! C@ @ +! 2DUP DUP SWAP DROP OVER LIT EXECUTE

NOTES

NOTES

NOTES

NOTES

Published by Interceptor Ltd.
Mercury House, Calleva Park Ind Est.
Aldermaston, Berks.
Telephone:(07356) 71145